Evaluation of Behavior-Driven Development

Version of October 29, 2012

John Horn Lopes

Evaluation of Behavior-Driven Development

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

John Horn Lopes
born in Huntington Beach, CA, USA

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Evaluation of Behavior-Driven Development

Author: John Horn Lopes
Student id: 1030299
Email: `j.a.hornlopes@student.tudelft.nl`

Abstract

Behavior-Driven Development is a recent addition to the family of Agile software engineering methods; the software engineering process of Behavior-Driven Development has not yet been extensively documented. We have therefore created a structured description of this process based on literature, and applied the process in a case study to evaluate if it provides stakeholders with enough information to successfully complete a project. The results of this evaluation show us a number of issues with the existing process. We suggest additions and clarifications to mitigate these issues and evaluate these propositions in the second part of the case study. This shows us that most evaluated changes are an improvement to the process: a more complete software engineering process for Behavior-Driven Development is achieved by incorporating our suggestions.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. phil. H-G Gross, Faculty EEMCS, TU Delft
Committee Member:	Dr. S. O. Dulman, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

Software plays a huge and ever increasing role in our lives. As software becomes more widespread both the number of software projects and the complexity of these projects increases. Not all projects are a success, and lessons learned from both successful and failed projects have lead to a number of software engineering methods. By providing processes to guide the stakeholders of a project, software methods attempt to increase the likelihood of success.

One group of software engineering methods are Agile methods, which follow the principles described in the Agile Manifesto [4]. Behavior-Driven Development (BDD) [26] is a recent addition to the family of Agile methods. BDD was first introduced as a different name for Test-Driven Development, and was transformed into a more complete method by adopting concepts from other processes and practices. Software engineers seem interested: a number of tools have been developed to support BDD activities [28]. The BDD software engineering process has been described in [15], however this informal description is unstructured. The goal of the research outlined in this thesis is to provide a complete, structured description of the process and evaluate whether BDD provides enough information to complete a software project successfully.

1.1 Problem statement

To guide our research we pose a number of questions:

1. *What is the software engineering process of Behavior-Driven Development?*

   We gather and organize information from available literature that covers Behavior-Driven Development to get an overview of the software engineering process. This results in a structured description of the process with a clear set of artifacts, roles and activities, which has been documented in chapter 3.

2. *Does the process provide sufficient information to complete a project?*

   To answer this question we need to analyze a project that follows the software engineering process of Behavior-Driven Development as we have outlined it in response

to question 1. We therefore conduct a case study by applying the process to a project in the domain of maritime safety and security, for a fictitious coast guard center. During the project we identify situations where the process does not provide enough information by continuously asking the following questions:

a) Is it clear what artifacts look like, and do the artifacts provide enough information for roles to perform the activities?

b) Are the roles described accurately?

c) Is it clear what each activity encompasses, and do roles have sufficient information to perform the activities?

The process of Behavior-Driven Development is iterative, and the project involves a project inception phase and seven iterations. In chapter 4 we describe the issues with the artifacts, roles and activities of the process that we identified for each iteration individually. Although there are a number of different roles in the BDD process, only two people are involved in the project: one person plays the role of customer, the other takes on the other roles. The source code of the system that resulted from the project is available on GitHub [8].

3. *If the process does not provide sufficient information, how can we clarify or extend the process description?*

   We use the lessons learned during the case study to suggest how the process can be changed to mitigate these issues. In chapter 5 we outline these suggestions in terms of additional or modified artifacts, roles and activities.

   To get an indication of the impact of our suggestions we then continue the case study for one iteration, using the updated process. We describe our experiences during this second part of the case study in chapter 6.

1.2 Thesis outline

The questions posed in the problem statement in section 1.1 guided the research outlined in this thesis. We start by describing the main concepts of Behavior-Driven Development in chapter 2. In chapter 3 we outline the software engineering process of Behavior-Driven Development, as it can be distilled from literature.

In chapter 4 we describe our experiences during the case study and the lessons we learned from those experiences. In chapter 5 we describe how these lessons lead to changes in the process in the form of additional artifacts, roles and activities. Where possible we used Test Driven Development, Acceptance Test Driven Development, Domain Driven Design or other agile processes as a source of information for these additions.

For a preliminary evaluation of the modified process we continued our case study for one more iteration to gauge the impact of our modifications. Our experiences and lessons learned during this final iteration are outlined in chapter 6. In chapter 7 we discuss the

results of our research in the context of the questions stated in 1.1, and the threats to validity. Finally, in chapter 8 we summarize our conclusions and propose directions for future research.

Chapter 2

Behavior-Driven Development

Behavior-Driven Development (BDD) [26] is an Agile development method that focusses on describing the behavior of the system from the perspective of its stakeholders [15], at all levels of granularity. BDD promises that this focus leads to improved communication and more value for stakeholders when compared to other Agile development methods.

BDD is based on Test-Driven Development, Acceptance Test-Driven Development and Domain-Driven Design. Before we can describe BDD we need to understand the concepts it is based on. We therefore give a short description in the next few sections, after which we will describe Behavior-Driven Development and how it combines these concepts in section 2.4.

2.1 Test Driven Development

Test-Driven Development (TDD) [13] is a technique for the design and implementation of software. TDD was first described as part of the Extreme Programming [14] software engineering process, and aims to encourage better software design. With TDD the architecture of the system is not determined up-front; the developer expands and implements the design in short iterations. Each iteration consists of three activities:

1. *Create unit test*
   First, the developer creates a unit test that can show whether the functionality is implemented in the system in a way that complies with its requirement. This forces the developer to think about what it means to correctly implement the functionality. In a way the unit test functions as a design specification. If the test is executed it will fail, since there is no implementation yet.

2. *Implement*
   The next step is to implement the functionality in the simplest way possible. The developer focuses only on the changed functionality, and does not yet take into account the design as part of the larger system. The developer is done with the implementation when it successfully passes the unit test.

3. *Refactor*
   In the final step the developer takes a step back and looks at what the implementation changes mean for the system as a whole. The changes may have introduced inconsistencies or duplication of functionality. The developer can improve the design by updating the system without affecting its functionality, an activity referred to as *refactoring*. Refactoring is enabled by the unit tests created during the first activity; the developer can change the system and the unit tests will give confidence that the system still functions correctly.

These activities are repeated with pieces of functionality until all required changes are implemented. Each iteration is very short, typically measured in minutes [24][25], the functionality created during each iteration is therefore also small. The tests created in the first step drive the design and implementation; this gives TDD its name.

TDD does not inherently lead to a top-down or bottom-up design strategy [13]: it depends on which test case the developer chooses to start with. The developer can start with a test representing a simple case of the entire system for a top-down design strategy. As the developer works out the necessary detailed behavior, more low-level classes are added to support this behavior. The developer can use a bottom-up strategy by starting with tests for low-level classes. As more of these classes are added, abstractions are surfaced through refactoring and form the higher level design.

The activities described above show that design, implementation and testing all take place during each iteration. However, the activities do not always cover all the design and testing needs of a software project:

- TDD is not always sufficient to create an adequate architecture. In [14] it is suggested that when developers feel that there are problems with the architecture as it has evolved during the TDD cycles, they have a meeting to investigate how they can improve the design. Small refactorings over a longer period of time can then be used to introduce the required design changes into the system.

- Most systems require additional testing activities. Although the developers design, implement and run unit tests, integration- and acceptance tests are also required. These testing activities will have to be performed outside TDD.

Evaluations of TDD [19][22][24] suggest that in general the quality of the system in term of the density of defects improves, although the required effort often increases. A study described in [23] suggests that applying TDD improves the design of a system by stimulating smaller, less complex units.

2.2 Acceptance Test-Driven Development

Acceptance Test-Driven Development (ATDD) [24][25] is a software requirements specification and verification process. ATDD emphasizes automation of acceptance tests and the specification of customer-readable requirements through concrete examples, which is also

referred to as specification by example [10]. The reasoning is that automated acceptance tests keep all participants of the process focussed on the goals of the software project, while the customer-readable requirements and acceptance tests improve communication.

The ATDD process [16][25] involves three roles: the customer, tester and developer. The artifact of the process is a set of executable acceptance tests. The process involves three activities, performed in order:

1. *Write acceptance tests*
   A tester will work with the customer to find tests that accurately outline the expected behavior corresponding to a requirement. All variations of the behavior are specified through concrete examples with clearly defined input and output, and are added to the specification as acceptance criteria. The customer must understand the documented tests, which often limits the notation to tables of example data [10] or Domain Specific Languages that resemble natural language [15].

2. *Automate acceptance tests*
   Although the tests include concrete examples they usually can not be used directly as test oracles. A developer will work with a tester to map concepts used in the examples to their implementation in the system, often by writing code. The developer can automate all examples for a requirement at once, or one at a time as the required features are implemented.

3. *Implementation*
   The developer designs and implements the functionality that makes the system meet the requirement, and is done when the system passes the acceptance test. The precise processes and techniques that are used implement the functionality are outside the scope of ATDD.

When ATDD is used as part of an iterative software engineering process the ATDD process can also be applied iteratively. Acceptance tests can first be written for all requirements of an iteration, after which the tests can be automated and implemented one by one.

The effectiveness of ATDD has not yet been extensively evaluated, although some success has been reported in a case study [11].

2.3 Domain-Driven Design

Users will use a software system to support their activities in a specific subject area, which is referred to as the *domain* of the system. Software teams need knowledge of the domain of the system to understand the motivation and requirements of the system and communicate efficiently with users and others outside the team. Domain knowledge is gathered by reading literature and interacting with users and domain experts. The results can be documented in a domain model, which includes the concepts and details relevant to the system.

Different domain models can be used throughout the software engineering process. An analysis model is created by analyzing and organizing the domain, with the goal of understanding the domain. When the system is designed, understanding of the domain is no

longer the primary goal. Other abstractions are appropriate, which leads to a design model. The use of different domain models complicates software development, since the knowledge captured in the models has to be kept synchronized.

Domain-Driven Design (DDD) [18] is a design philosophy that attempts to solve this problem by using a single domain model throughout the project. The model captures entities, activities and rules of the domain and is created and maintained by the software team in collaboration with domain experts. Frequent collaboration between the software engineering team and domain experts leads to a common understanding of the domain model: the team will gain knowledge of the domain while experts will learn how to express themselves in terms of the model.

2.3.1 Uses of the domain model

Evans [18] suggests three uses for the domain model in a software project: to capture knowledge of the domain, to function as a common language and as part of the design of the system. In the following subsections we will describe these uses in more detail.

Distilled knowledge

As the project progresses domain experts and the software development team continuously collaborate to improve the domain model. Domain experts learn how to express themselves concisely in terms of the domain model and the model helps them discover vagueness or contradictions in their thinking. The software team gains a better understanding of the domain and discovers better ways to express the domain in the domain model. The result is that domain experts and the software development team distill their knowledge of the domain to the essentials necessary for the project.

Ubiquitous language

A successful software project requires good communication, which in turn relies on a shared language. Domain experts think and reason in terms of their domain language. Developers do the same, using concepts from the domain of software development. Analysts and developers translate between these domains, mapping domain concepts to design. However, information can be lost in this translation, which causes different people to have different interpretations of concepts.

In a project with a ubiquitous language domain experts and the software team communicate in a common language, both in conversation and in documents, which means no translation is necessary. [18] suggests the use of the domain model as the basis for such a language. The vocabulary of the language includes the names of classes and important operations and includes terms that make it possible to reason about business rules and activities. The language is used in both communication and the design of the system.

The language is created in tandem with the model: when one changes so does the other. When the model is updated, all documents written in the language are also updated. Since the updated language allows for more precise communication, inconsistencies in the model are discovered and clarified.

Binding model and implementation

It is possible to create the domain model during analysis and then use it to design the system, creating a design model in the process. However, as discoveries are made during the design phase the domain model may no longer be adequate since it lacks certain details. There is a risk that part of the domain model is then abandoned with the result that not all knowledge acquired during analysis is retained.

DDD therefore requires the use of a domain model that also accurately describes the domain-specific part of the design. The premise is that both the domain and the design can be modeled in many different ways. By searching out a model that accurately describes both the domain and the design, the concepts from the domain always map directly to similar concepts in the design. This is referred to as Model-Driven Design. Like the ubiquitous language, the design is created in tandem with the domain model: when the model is updated the design and thus the code is updated, and vice versa.

2.3.2 A Domain-Driven Design

Domain-Driven Design [18] is a design philosophy for the domain-specific part of the design. DDD describes how the domain model can be mapped to a design and how the design can be updated and refactored throughout a project. To facilitate changes [18] recommends a layered architecture with a separate layer representing the domain model. By ensuring this layer is loosely coupled to other layers, changes to the domain model will have minimal impact on the rest of the design.

Concepts from the domain model are mapped to objects, of which [18] distinguishes the following types:

- *Entities*: objects that represent concepts with a unique identity for each instance
- *Value objects*: objects that represent values without a unique identity
- *Associations*: relationships between objects
- *Services*: domain operations that are not directly tied to specific entities or value objects
- *Modules*: groupings of objects that represent related concepts

Evans [18] also outlines a number of object types that are used to manage the life cycles of objects, allowing the system to create, persist, retrieve and remove domain objects.

2.4 Behavior-Driven Development

Dan North introduced Behavior-Driven Development [26] as a different name for Test-Driven Development after observing that Test-Driven Development was often perceived as a testing technique [12][23]. By replacing the word "test" in the name of TDD with "behavior", BDD aimed to emphasize that TDD is about design, not testing.

Over time BDD has grown to become a more complete method [15][27] by absorbing concepts from ATDD and DDD, which has lead to the definition of the three principles of BDD:

- *Enough is enough*: we should do just enough up-front analysis and design to get us started, any more is wasted effort.
- *Deliver stakeholder value*: we should only do what delivers value or increases our ability to deliver value.
- *It is all behavior*: at each level of granularity we can use the same language to describe the behavior of the system.

The requirements of the system are expressed as user stories [14][16] and extended with acceptance criteria in the form of scenarios. These scenarios are then automated where feasible to form the automated acceptance tests required for ATDD. The design and implementation of the functionality required by stories takes place in TDD cycles.

BDD has adopted the concept of a ubiquitous language from DDD [15]. Although DDD uses the domain model in other ways and is primarily a design technique, BDD has not been explicitly associated with these other aspects of DDD.

In this chapter we have provided on a high-level overview of BDD and its origins: Test-Driven Development, Acceptance Test-Driven Development and Domain Driven Design. In the next chapter we will take a closer look at the process that is followed when using the BDD method for a software project.

Chapter 3

A Behavior-Driven Software Engineering Process

The most extensive description of the software engineering process of Behavior-Driven Development is given in a book [15] co-authored by BDD creator Dan North. However this description is somewhat unstructured: it does not distinguish discrete activities and roles. In this chapter we give a more structured description.

We separate the process into the phases common in software engineering processes: requirements, design, implementation and testing. BDD is an iterative process, and a BDD project goes through these phases during each iteration. However, before the first iteration starts some preparation is done [15], which we will group inside a project inception phase. We also distinguish what the activities, roles and artifacts are for each of these phases:

- *Activities*
  The activities of the process are the tasks that people participating in the process perform.

- *Roles*
  Each person participating in the process does so in one or more roles, and roles have certain activities associated with them. Each role has to be performed by at least one person; if a person is not available during part of the project, someone else will have to temporarily take on the role.

- *Artifacts*
  The output of a software engineering project is the system itself, usually in the form of its source code. However, during the activities the roles create other documents that support the process. We will refer to these documents as the artifacts of the process.

3.1 Project Inception

The first step in a project is determining the motivation for the project by creating a vision statement. Before iterative development can start the initial requirements are also determined.

3.1.1 Artifacts

Vision statement

The vision statement is a short natural language statement describing the motivation for a system in the form of a vision. The statement is extended with the outcomes that will help achieve this vision. The vision statement functions as a reference for roles throughout the project and helps them stay focussed on the high-level goals. For example, the vision statement we will work with in the case study is:

```
Vision
Improve maritime safety and security

Outcomes
- Real-time overview of vessel locations
- Easy access to vessel information
```

3.1.2 Roles

Analyst

The analyst helps the customer with the definition of a vision statement.

3.1.3 Activities

Define vision and outcomes

A software project starts with a customer that has an idea of how a system can provide value. This business value forms the vision; the means by which the system will achieve the value are the project outcomes. The analyst helps the customer express the vision and outcomes as a vision statement.

Define initial requirements

Before the first iteration is started, an initial set of requirements is needed. The details of gathering requirements are outlined in the requirements phase in section 3.2.

3.2 Requirements

The requirements phase at the beginning of each iteration allows the customer to determine which behavior has to be added to the system. This phase is usually ahead by one iteration and involves the preparation of requirements for the *next* iteration. During the project inception the initial requirements are established; during iterative development the customer can add or change requirements, and add more detail to existing requirements.

3.2.1 Artifacts

Feature

The high-level requirements of the system are expressed as features [15]. Each feature is a short natural language statement using the ubiquitous domain language (as discussed in 2.3.1) and expresses the who, what and why of each requirement. A common template for a feature is:

Feature: *Name*	
In order to	*business value*
As a	*user role*
I want	*behavior*

User roles represent the different types of users of the system, and always represent people, not external systems: interfacing with external systems is always done to support a feature required by specific user roles.

Features express the user role that needs a certain behavior of the system, the behavior itself and the motivation in the form of the business value. Each feature has a unique name so that it can be easily referenced. Since each feature has to present a direct business value, the behavior can not be restricted to certain layers of the architecture or have a purely technical goal. One of the features we will work with in the case study is the *Map View* feature, which describes that vessels in a certain area will be shown on a map:

Feature: *Map View*	
In order to	*assess the situation in my area*
As a	*coast guard*
I want	*to see the location of each vessel marked on a map*

Story

Features are sometimes too large to estimate accurately and complete within one iteration. They are therefore broken down into stories [15][16]. Each story specifies a part of the scope of the original feature and express the who, what and why of a requirement using the same template. While stories are more detailed than features, the goal is not to capture all details of the requirement. Stories also have to present a direct business value and therefore cut through the whole architecture.

Scenario

Scenarios [15][16] specify the acceptance criteria associated with the requirements expressed in stories. Each scenario gives concrete examples of the behavior described in a story and helps refine the scope of the story. Scenarios become a part of the story they are associated with by appended them to the story text using the template:

Scenario: *Name*	
Given	*state*
When	*action*
Then	*outcome*

The name gives a short description of the scenario. The state describes the assumptions about the state of the system at the beginning of the example. Action describes the action the user role performs. Finally, outcome describes the expected outcome which could be an output or state of the system. The whole scenario is written in a Domain Specific Language (DSL) similar to natural language, which makes it possible to automate the scenario to form the acceptance tests while keeping it easy to read and understand for customers. For example, one of the scenarios of the *Map View* feature will describe a vessel in an area of the map:

Scenario: *show vessel inside map area*	
Given	*vessel "Seal" at position "52.01N, 3.99E"*
When	*I view the map area between "52.10N, 3.90E" and "51.90N, 4.10E"*
Then	*I should see vessel "Seal" at position "52.01N, 3.99E"*

3.2.2 Roles

Analyst

The analyst is responsible for finding features and user stories that describe the requirements. While gathering these requirements the analyst will work closely with the customer.

Tester

The tester defines the scope of each story by adding acceptance criteria in the form of scenarios, and splits stories with a large scope into multiple stories when necessary.

3.2.3 Activities

Find features

The analyst first identifies feature sets, general areas of functionality required to deliver the outcomes defined in the vision statement. Examples of feature sets are "inventory management" and "reporting". Within the feature sets the analyst will identify the users of the system, and generalize these users into user roles. Finally, the analyst will determine how each user role uses the system, and with what goal. These uses of the system are described as features. The analyst should attempt to keep the scope of each feature limited while ensuring that the feature still delivers direct value to a stakeholder.

Often there are also what could be considered non-functional requirements. In BDD all requirements are considered to be functional for some stakeholder of the system. By including all stakeholders as user roles, all functionality can be described through features. For example, a requirement for a system that shows vessels on a map could specify that

OpenStreetMap [7] map data should be used to prevent licensing fees. The source of map data is non-functional to the primary users, however it is a functional requirement to the legal team, and can be described as a feature.

Find stories

The scope of a feature may be too large for the feature to be implemented during a single iteration. Such features are broken down into separate stories, where each story delivers part of the value. Features that already have limited scope can be directly used as stories.

When decomposing a feature into stories the analyst will try to make the feature less abstract by finding what different tasks the user may perform in the context of a specific feature. These different tasks are candidates for user stories.

When an feature is not a candidate for implementation during the next few iterations, it does not need to be broken down into stories. This can be done during a later iteration, just before the feature has to be implemented.

Prioritize stories

The analyst will, with help from the customer, prioritize the stories, ordering the stories from high to low business value. When a high risk is associated with a story it can also be moved up, which allows the customer to discover whether the risk will turn out to be a real problem during early iterations. The implementation of stories is likely to follow the order that the customer sets in this activity, however developers can suggest to give certain stories a higher priority for architectural reasons.

Find scenarios

The behavior described in stories is likely to be abstract. With input from the customer and analyst, the tester determines concrete (i.e. with actual data) examples of the story which will be described as scenarios. One scenario will usually describe behavior of the system as a user is performing valid actions, with valid input data: the normal path. Exceptional paths, for example where the user inputs invalid data, are also described as scenarios.

During the creation of scenarios the tester may find that a certain story has many scenarios. This may be a sign that the scope of the story is too large, and the tester may decide to split a story into a number of separate stories.

Scenarios are only added to stories which are candidates for implementation during the next few iterations. Adding detail to stories with a low priority could be wasted effort; the details may never be used as requirements change.

3.3 Design

When the developers and testers know *what* has to be implemented, the next step is to determine *how* it can be implemented: the design phase. BDD relies on TDD for all design activities: developers will create the architecture of the system while writing unit tests and

refactoring. Since TDD relies on short iterations of design and implementation, we will treat the related activities with the implementation phase.

3.4 Implementation

During the implementation phase the behavior specified in stories is added to the system by implementing scenarios.

3.4.1 Artifacts

Acceptance tests

Scenarios indicating acceptance criteria are attached to each story, and acceptance tests are used to verify whether these criteria have been met. Since the scenarios are written in a DSL, the acceptance tests can be created by making the scenarios themselves executable. When the tests pass, the team knows the story has been implemented correctly.

Unit test

Unit tests are low-level tests that verify the behavior of individual units, often classes in object-oriented programming environments.

3.4.2 Roles

Developer

The developer is responsible for the automation of acceptance tests and the design and implementation required to change the behavior of the system as required by the stories.

3.4.3 Activities

Make acceptance test executable

Scenarios provide placeholders *state*, *action* and *outcome*, and the values of these placeholders provide the start state, input and expected output and end state for acceptance test, respectively. However, the placeholder values depend on the domain and the type of system being developed. To enable the use the scenarios as executable acceptance tests the developer maps the placeholder values to program code that interacts with the system with the help of the tester. After the mapping has been created tools can be used to interpret and execute the scenarios, and each scenario provides an executable acceptance test. The activity can be simplified by the application of Domain-Driven Design, since domain concepts used in scenarios then map directly to objects in the domain layer of the system.

Stories and their scenarios describe behavior of the system as a whole and not specific architectural parts. Executable acceptance test therefore ideally exercise all parts of the architecture that are involved in providing the behavior. As a result, BDD acceptance tests are generally driven through the user interface. When this is not feasible it is recommended

[15][25] to selectively automate tests using the user interface and drive the tests for the remaining scenarios from a lower level.

Design and implement scenario

The developer applies the Test Driven Development (TDD) technique to implement the behavior required by a scenario. In section 2.1 we noted that TDD is not tied to a specific design strategy, and described how top-down and bottom-up strategies can be executed with TDD.

The use of Acceptance Test-Driven Development leads to the use of a different design strategy in BDD. The acceptance tests interact with the system using its external interfaces; being driven by acceptance tests therefore implies that design and implementation should start at these interfaces. This design strategy is referred to as *outside-in* [15]: the developer starts designing the system from the outside, an interface of the system, and works inward.

The first TDD cycle starts with a test for the unit that will implement the interface. When the developer adds the behavior behind the interface, much of this behavior is deferred to other objects. These collaborating objects may not have been implemented yet and the developer uses mock objects for these collaborators in the unit test. When the initial unit has been implemented, the developer proceeds by designing and implementing each collaborator, again using mock objects to support the unit tests. This procedure is applied recursively until all collaborators have been implemented. This variation of the TDD technique is sometimes referred to as *mockist TDD* [5] due to its extensive use of mock objects.

With an outside-in design strategy every object is designed only to support the behavior at the interface of the system. The motivation for its use in BDD is that this leads to the simplest architecture that can support the required behavior.

After the design and implementation of the system have been updated to support the scenario, the developer will run the automated acceptance tests to see if the scenario has been correctly implemented. Since the unit tests use mock objects rather than the real collaborators, the integration of the units has not been tested. The acceptance tests therefore also function as integration tests.

This activity is repeated for all scenarios of a story. When all the scenarios have been implemented and pass their acceptance test, the developer moves on to the next story.

3.5 Testing

As we've seen in the previous section, a certain amount of testing takes place during the implementation phase. The testers and developers have worked together to create executable acceptance tests, and developers run these tests before reporting a story as implemented. Other testing activities are not described in [15] or other BDD literature.

3.6 Summarizing the process

We have seen that the behavior driven software engineering process specifies a number of artifacts, roles and activities, which we have listed in table 3.6. Although the design and

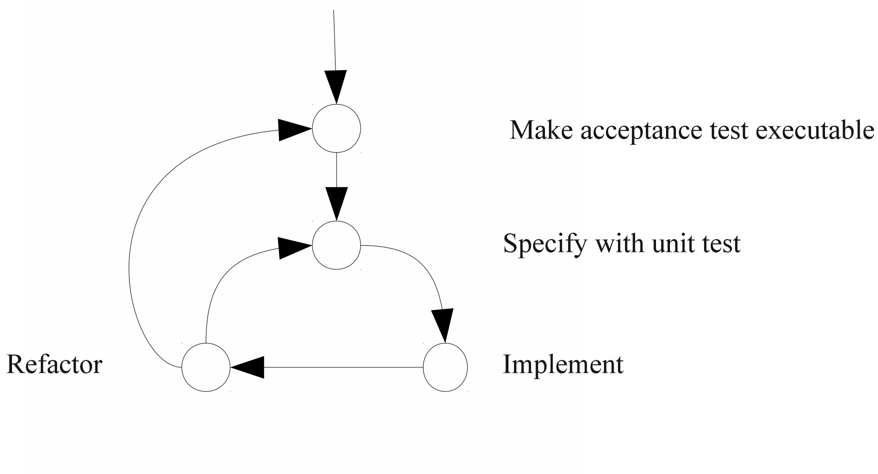


Figure 3.1: The BDD implementation cycle uses the scenarios created during the requirements phase to create an executable acceptance test, which in turn drives implementation using TDD. The TDD cycle starts with the creation of a unit test, after which the change is implemented and the system is refactored. When the scenario is fully implemented the developer continues with the next scenario.

Table 3.1: An overview of the BDD process listing the artifacts, roles and activities.

Project Inception	
Artifacts:	Vision Statement
Roles:	Analyst, Tester
Activities:	Define vision and outcomes, Define initial requirements
Requirements	
Artifacts:	Feature, Story, Scenario
Roles:	Analyst, Tester
Activities:	Find features, Find stories, Prioritize stories, Find scenarios
Implementation	
Artifacts:	Acceptance test, Unit test
Roles:	Developer
Activities:	Make acceptance test executable, Design and implement scenario

testing phases are common in software engineering processes, BDD does not specify any activities for those phases. Design and testing are part of the TDD cycle performed during the implementation phase instead.

3.7 Tools

Now that we have described the BDD software engineering process and its activities, we will look at some of the tools that can support these activities.

3.7.1 Project inception

The *Define vision and outcomes* activity is a conversation between the analyst and the customer, and the results are documented in a natural language document, the *vision statement* artifact, therefore no tools are necessary.

3.7.2 Requirements

The activities in the requirements phase also revolve around conversations between the team and the customer, the results of which are documented in *feature*, *story* and *scenario* artifacts. These artifacts are mostly natural text, with some restrictions are imposed by their format. For features and stories this format is a matter of convention to ensure they can be easily understood by all participants in the project. Due to the simplicity of the format a text editor is sufficient to create and edit features and stories.

Scenarios have some stronger restrictions since they represent the acceptance tests that will later have to be made executable with the *Make acceptance test executable* activity. A Domain Specific Language (DSL) is used, designed to be both easy to understand by the customer and easy to interpret programmatically. The most popular DSL for this purpose is Gherkin, which we will discuss in section 4.2.1. Although a specialized editor can help work with DSLs in general, the Gherkin language can be written with a text editor.

3.7.3 Implementation

The implementation phase involves more technical aspects, there will therefore be more opportunities to support activities with tools.

Acceptance testing

We described how the *Make acceptance tests executable* activity maps parts of scenarios to program code, after which this mapping can be used to execute scenarios automatically. The automated execution of scenarios therefore requires specialized tools for acceptance testing. Using the scenarios and the mapping as input, these tools will interpret each scenario, apply the mapping and execute the associated program code. The interpretation of scenarios requires that the scenarios are written in a DSL. We describe Gherkin, a DSL used by several tools [1][2][3] for this purpose, in 4.2.1. Since the DSL depends on the acceptance testing tool, the choice of acceptance testing tool influences the way scenarios are written during the requirements phase of the process.

Unit testing

The TDD cycle of the *Design and implement scenario* activity depends on the use of a unit test tool. Many unit test tools are available, most of which use the test-centric terminology of jUnit-style tests: developers write test cases and test methods, and use assertions to verify behavior. As we described in section 2.4, the use of the word *Test* in Test-Driven Development was cause of confusion, which motivated the introduction of BDD. Therefore we would prefer a tool that performs the same functions yet uses language constructs that focus more on design aspects. Such BDD-style unit test tools are available for some platforms, and allow the developer to write *specs* and *expectations* instead of test cases and assertions, respectively.

Chapter 4

Case study

In the problem statement in section 1.1 we described the setup of a case study aimed at evaluating the BDD software engineering process. We used this setup to perform a case study with a project in the domain of maritime safety and security. In this chapter we describe the experiences of the involved process roles for each activity described in chapter 3. We will start with the project inception and then go through each iteration, describing how each activity was performed and what difficulties the roles experienced in terms of question 2 of 1.1. In the section 4.11 we will outline general observations not tied to a specific iteration.

Every project needs a project management process that facilitates estimation, prioritization and planning. BDD does not offer these facilities, however as an agile process it is compatible with agile project management processes. We have chosen to use Scrum for our case study, since it is well documented and is commonly used with features and stories as requirements specifications [16]. There are a number of different roles in the BDD process, which are taken on by two people in this project: one person plays the role of customer, the other takes on the other roles.

We now describe the domain of the project in more detail in section 4.1 to clarify the examples of artifacts we provide later in this chapter. We discuss the tools we used during our project, in section 4.2.

4.1 Domain of the project

The case study involves a project in the domain of maritime safety and security. As the requirements are gathered it becomes clear that the system also includes concepts from the domain of Geographical Information Systems (GIS). Vessels are shown on a map, where each vessel is represented by a marker. The icon used for each marker also conveys information like the type of vessel and its heading, while clicking an icon shows a pop-up window with detailed information. The source of vessel information is the Automated Identification System (AIS). The source code of the system that resulted from the project is available on GitHub [8].

In the remainder of this section we will describe those aspects of the Automated Identification System and Geographical Information Systems that are relevant to the case study project.

4.1.1 The Automated Identification System

The AIS is a vessel tracking system introduced by the International Maritime Organization [20] to improve the safety of navigation. Vessels use VHF transceivers [21] to communicate with each other, shore base stations and satellites [17] to exchange vessel information:

- Static information, e.g. the name and type of vessels
- Dynamic information, e.g. the position, speed, heading and course of vessels
- Voyage information, e.g. the destination of vessels

The availability of this information allows for more reliable collision avoidance, more detailed information for authorities and improved traffic management.

The AIS distinguishes two classes of AIS equipment: class A and class B. Larger vessels and commercial passenger vessels are required to carry class A equipment; class B equipment is used on smaller, non-commercial vessels, and its use is not mandatory in most waters. Both types of equipment use the same communication protocols and are able to receive messages from the other type of equipment, however they send different messages. Messages sent by class A equipment generally have more detailed information and are sent more frequently.

4.1.2 Geographical Information Systems

The project also involves a number of concepts from the domain of Geographical Information Systems (GIS). In a GIS a map consists of different layers, each with a different type of information. While one layer may be a geographical map, layers on top may be used to position different types of markers representing vessels or navigational aids like buoys.

4.2 Tools

In section 3.7 we described the types of tools necessary to support the activities of the BDD process. In this section we will look at the specific tools used for the case study.

The initial requirements for the project showed that the main language for the project would be Ruby. After some research the developer selected the Cucumber and RSpec tools, since they are well documented [15], actively developed and use BDD-style syntax for describing behavior. Cucumber combines the popular Gherkin DSL with Ruby for automated acceptance tests; RSpec allows us to write BDD-style unit tests for Ruby code. We will describe Cucumber in section 4.2.1 and RSpec in section 4.2.2.

During the first iteration of the project the developers chose to use a Javascript-based web interface as the user interface of the system. The developer chose to use Jasmine as the

Listing 4.1: Example of a Gherkin feature file with a feature and one associated scenario

```
Feature: Vessel Details
  In order to get information about a vessel
  As a coast guard
  I want to be able to select a vessel and see its details

  Scenario: class A vessel
    Given vessel "Sea Lion" with details:
        | MMSI     | 245000000            |
        | Class    | A                    |
        | Type     | Cargo                |
        | Status   | Underway using engine |
        | Position | 51.99N, 4.05E        |
        | Heading  | 290                  |
        | Speed    | 13.1                 |
     When "Sea Lion" sends a position report
      And "Sea Lion" sends a voyage report
      And I select vessel "Sea Lion" on the map
     Then I should see all details of vessel "Sea Lion"
```

unit test tool for Javascript, since it is the only actively developed BDD-style unit test tool and is well documented. Jasmine will be described in 4.2.3.

4.2.1 Cucumber

In section 3.7.3 we outlined how the *Make acceptance tests executable* activity is supported by an acceptance testing tool. Cucumber [3][15] is such a tool; use of Cucumber consists of three parts. First, scenarios are written in the *Gherkin* DSL, which allows Cucumber to interpret the scenarios. Second, the developer associates a block of Ruby code with each step of the scenarios using *step definitions*. Finally, the Cucumber command-line tool combines the features written in Gherkin and the step definitions to execute each scenario as an acceptance test. In the following subsections we will describe these three aspects of Cucumber in more detail.

Gherkin

Cucumber requires features to be written in a DSL called Gherkin. Although Gherkin was originally developed specifically for Cucumber, it is now used by other BDD acceptance testing tools, such as Behat [1] and Behave [2].

Gherkin requires each feature to be saved in a separate file; an example is provided in listing 4.1. The file starts with the *Feature* keyword, followed by the name of the feature. On the next few lines the feature text is specified. This text has no restrictions other than its indentation, however both [3] and [15] suggest using the format described in section 3.2.

The rest of the feature file is used to specify the scenarios associated with the feature. A scenario starts with the *Scenario* keyword followed by a short description of the scenario. The following lines provide the *Given*, *When* and *Then* clauses of the scenario, which we

described in section 3.2. Each clause can consist of multiple steps, where each line starting with the *And* keyword indicates a new step. For example, the *And* keywords on lines 16 and 17 of listing 4.1 indicate steps that are part of the *When* clause started on line 15. The lines clarify that the *When* clause consists of a number of discrete steps and each line can be independently reused in other scenarios.

The example in listing 4.1 also shows how double-quoted strings are used to include example data in a step. Cucumber will pass the values in double-quoted strings to step definitions as a string, which enables developers to access the values. Data tables offer another way to include example data, and are defined using the pipe symbol | keyword. Each table is passed to the step definitions as an object.

The example in listing 4.1 shows that Gherkin is very similar to natural language. In our examples we used keywords in English, however Cucumber supports over 30 languages [3]. The use of a DSL close to natural languages allows the customer to read -and perhaps even write- stories and scenarios. In fact, the examples of feature and scenario artifacts we described in section 3.2 were already written in Gherkin.

Step definitions

Step definitions are used to map the behavior described in scenarios to the implementation of that behavior in the system: each scenario step can be mapped to a block of Ruby code. An example of such a mapping for the *Vessel Details* feature (listing 4.1) is shown in listing 4.2.

For each distinct step of the scenarios the developer creates a step definition, which associates a regular expression that matches the step with a block of code. Matched groups inside the regular expression are passed into the block of code, as block arguments. For example, on line 1 of listing 4.2 we see how the vessel name is matched by the group `(.*?)` of the regular expression and is passed to the block of code in the variable `name`. If a table is as part of a step, it is passed as the last block argument; the table defined on lines 8-14 of listing 4.1 is passed in the variable `table` on line 1 of listing 4.2.

Each step definition has its own scope. State that needs to be maintained between different steps therefore has to be saved in instance variables (prefixed by @ in Ruby). The values of these variables will be available to all step definitions.

An acceptance test is not complete without a test oracle that compares the output and state of the system to the expected output and state. The *Then* step defines the expectations its step definition is where expectations are compared to actual results. A step definition can signal failure by raising an exception. The developer can perform the comparison and raise an exception explicitly, as is done on line 32 of listing 4.2, or use more advanced matching or assertion capabilities provided by frameworks such as RSpec.

Command-line tool

The Cucumber command-line tool uses the feature files and step definitions to execute the acceptance test. As the command-line tool goes through each scenario, it tries to match each step against the regular expressions of the step definitions. When a match is found, the

Listing 4.2: Example of the five step definitions for the *Vessel Details* feature from listing 4.1. The details of some definitions has been removed for brevity

```
Given /^vessel "(.*?)" with details:$/ do |name, table|
  @fields = table.rows_hash

  if @fields['Class'] == 'A'
    vessel_class = Domain::Vessel::CLASS_A
  else
    vessel_class = Domain::Vessel::CLASS_B
  end

  @vessel = Domain::Vessel.new(@fields['MMSI'], vessel_class)
  @vessel.name =  name
  @vessel.heading = @fields['Heading'].to_f
  # And more assignments
end


When /^"(.*?)" sends a position report$/ do |name|
    # Ruby code
end

When /^"(.*?)" sends a voyage report$/ do |name|
    # Ruby code
end

When /^I select vessel "(.*?)" on the map$/ do |name|
    # Ruby code
end

Then /^I should see all details of vessel "(.*?)"$/ do |name|
  @fields.each do |_, value|
    if not page.has_content?(value)
      raise "Text '#{value}' not found on page"
    end
  end
end
```

Listing 4.3: Part of a RSpec specification for the `LatLon` class, which represents pairs of latitude/longitude coordinates

```
require 'spec_helper'

module Domain
  describe LatLon do
      it "can create a new LatLon from a string" do
        latlon = LatLon.from_str("47.16N, 9.66E")
        latlon.lat.should eq(47.16)
        latlon.lon.should eq(9.66)
      end

      it "rejects invalid string input" do
        expect { LatLon.from_str("1.1 N, 9.6E") }.to raise_error
      end

      it "can be converted to a string" do
        latlon = LatLon.new(47.16, -9.66666666)
        latlon.to_s.should eq("47.16N, 9.6667W")
      end

      it "can be compared to other LatLon objects" do
        l = LatLon.new(1.0, 2.5)

        l.should eq(LatLon.new(1.0, 2.5))
        l.should_not eq(LatLon.new(2.0, 2.5))
        l.should_not eq(LatLon.new(1.0, 2.0))
      end
  end
end
```

associated block of Ruby code is executed. If an exception is raised the tool displays the exception, marks the scenario as failed and continues with the next scenario. If no match is found for a step the tool will display an error message and suggest a regular expression that can be used for a new step definition. When all scenarios have been processed the tool displays a summary, including a list of the scenarios that failed.

4.2.2 RSpec

RSpec [15] is a BDD-style unit test tool for Ruby that uses *specs* and *expectations* instead of tests and assertions; an example of a spec file is given in listing 4.3. The keyword `describe` is used to indicate which module or class is being described by this specification. The block that follows contains properties of the described class, expected behavior is shown with examples. An example starts with the `it` keyword followed by a description and a block of code that specifies the detailed behavior. Rspec also uses the `describe` and `it` constructs to format the test output in a human-readable format, as is shown in listing 4.4.

RSpec uses *expectations* to verify behavior. Expectation are added to objects by calling its `should` or `should_not` method. RSpec automatically adds these methods to all objects,

Listing 4.4: Output of RSpec when running the example provided in 4.3

```
$ rspec spec/lib/domain/latlon_spec.rb

Domain::LatLon
  can create a new LatLon from a string
  rejects invalid string input
  can be converted to a string
  can be compared to other LatLon objects

Finished in 0.00493 seconds
4 examples, 0 failures
```

and are passed a *matcher* as an argument. The matcher is used to verify if the object meets some expectation, where the expectation is specific to the matcher. For example, on line 7 of 4.3, the call `eq(47.16)` returns a matcher that returns true when it is passed the value `47.16`, false in other cases. Each time `should` is called it will use the provided matcher to verify that the object matches the expectation.

Although RSpec introduces a number of keywords like `describe` and `it`, the spec files are plain Ruby, and the developer can therefore run the spec files directly using the Ruby interpreter. However, this becomes cumbersome as the project grows and more spec files are added. RSpec therefore also provides the `rspec` test runner, a command-line tool which will run all spec files in the `spec` subdirectory and summarize the results.

4.2.3 Jasmine

In the case study the developer used Javascript in addition to Ruby. Since RSpec specifications are written in Ruby, the developer could not use this tool for the specification of Javascript constructs. Jasmine offers similar functionality for Javascript, using a similar syntax (listing 4.5). It also uses `describe` and `it` constructs, in the form of special functions. The first argument is a description, the second argument is a function with the detailed specification. Jasmine will use the descriptions to format human-readable output, as is shown in figure 4.1.

In Jasmine the object for which an expectation is specified is wrapped in a call to the `expect` function. The `expect` call returns an object with a number of matcher methods. Matcher methods will raise an exception when they are called on an object that does not meet their expectation.

For example, consider line 5 of listing 4.5. `expect(latlon.lat)` returns a matcher object encapsulating `latlon.lat` and has a `toEqual` method (in addition to many other matcher methods). When `toEqual` method is called, the value passed will be compared to the object encapsulated by the matcher object, and an exception raised if they are not equal.

Since Jasmine specifications are written in Javascript, a Javascript runtime environment is required to run the specifications. Jasmine has an embedded web server, which allows any web browser to be used as a runtime environment. The developer connects to the web server to execute the specifications, after which the results are shown (figure 4.1).

Listing 4.5: Partial specification of the LatLon Javascript prototype

```
describe("LatLon", function() {
  it("holds the lat and lon coordinates", function() {
    var latlon = new LatLon(52.0, 4.0);
    expect(latlon.lat).toEqual(52.0);
    expect(latlon.lon).toEqual(4.0);
  });

  it("can check if it has equal values as another LonLat object",
      function() {
    var latlon1 = new LatLon(10,20);
    var latlon2 = new LatLon(10,20);
    var latlon3 = new LatLon(10,30);
    var latlon4 = new LatLon(20,20);
    var latlon5 = new LatLon(10.00005,20.00005);

    expect(latlon1.equals(latlon2)).toBeTruthy();
    expect(latlon2.equals(latlon1)).toBeTruthy();
    expect(latlon1.equals(latlon3)).toBeFalsy();
    expect(latlon1.equals(latlon4)).toBeFalsy();
    expect(latlon1.equals(latlon5)).toBeTruthy();
  });
});
```

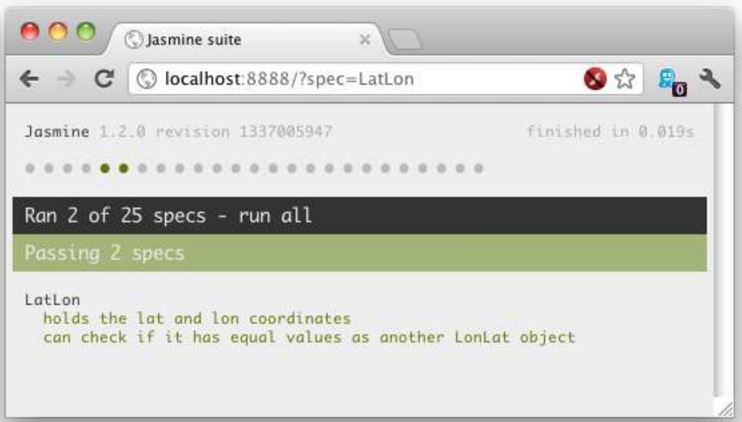


Figure 4.1: Jasmine has an embedded web server, which allows the developer to run specifications in any web browser.

4.3 Project Inception

4.3.1 Define vision and outcomes

When the analyst first discussed the project with the customer, the customer was focussing on what a solution would look like: showing up-to-date vessel information on a map. By asking why this was a desired solution, the analyst was able to determine the vision that the customer had for the system and the desired outcomes of the project. The resulting vision statement was documented in a plain text file:

```
Vision
Improve maritime safety and security

Outcomes
- Real-time overview of vessel locations
- Easy access to vessel information
```

The process was clear about how the analyst had to perform this activity, and the analyst had no problems creating the vision statement.

4.3.2 Define initial requirements

Find features

The first step toward finding features for the project was the identification of feature sets that would help achieve the outcomes defined in the vision statement. In this case there was only one feature set: the overview map. The analyst then proceeded by explaining the feature specification format and created the first few features with the input from the customer. After these first features the customer was able to add features with limited guidance and refinement by the analyst. This resulted in the set of initial features listed in table 4.3.2.

The features all require information about the vessels in a certain area, yet the stories themselves do not reveal the source of this information. The analyst discovered that the system would have to gather this information by processing messages from the Automatic Identification System (AIS), and that an existing system would provide a feed of raw AIS messages. The analyst had difficulty translating this requirement to a feature. One attempt was the *Vessel Information* feature:

```
In order to get up-to-date vessel information
As a coast guard
I want the vessel information to be updated with information from the AIS
```

Although it follows the feature format, this feature can not be implemented directly. We need vessel information to complete the other features, however we will not know exactly what information is required before these features are implemented. This interdependence can not be resolved without fully determining the requirements up front, a practice incompatible with BDD.

A related issue surfaced when the customer indicated that the system would have to be composed of different services communicating with using the ZeroMQ protocol [9].

Table 4.1: Initial features, displayed in order of decreasing initial priority. In this project the features are also used as stories.

Map View	In order to assess the situation in my area As a coast guard I want to see the location of each vessel marked on a map
Vessel Heading	In order to see the direction each vessel is moving in As a coast guard I want to to see the heading of each vessel marked on the map
Vessel Speed	In order to gauge the speed of vessels As a coast guard I want to see a line behind each marked vessel indicating speed
Vessel Class	In order to discriminate professional shipping and leisure shipping As a coast guard I want to see vessels of a specific class marked with a specific shape
Vessel Type	In order to distinguish vessels of different types As a coast guard I want to see vessels of a specific type marked with a specific color
Vessel Details	In order to get information about a vessel As a coast guard I want to be able to select a vessel and see its details
Vessel Movement	In order to see up the latest location for a vessel As a coast guard I want the vessel location to be automatically updated on the map

The motivation for this requirement was policy, not technical necessity. This requirement influenced how other features had to be implemented, and did not form a feature of itself.

These types of requirements cut across sets of features, and can therefore not be implemented as features themselves. Since there were only a few of these requirements for this project and only two people were involved, this was not an issue: they were easily remembered. However for a larger project or team a more structural approach is likely required.

Find stories

The analyst considered all of the features determined during the Find features activity to already have a sufficiently limited scope, therefore the features were directly used as stories.

Prioritize stories

The analyst and the customer determined that the most basic function of the system was showing the position of vessels on the map, since the other stories all depend on the system having this ability. The Map view story therefore received the highest priority. This procedure was repeated for the remaining stories, which resulted in table 4.3.2, where stories are listed in order of decreasing priority.

Listing 4.6: Initial scenarios of Map View story

```
Feature: Map View
  In order to asses the situation in my area
  As a coast guard
  I want to see the location of each vessel marked on a map

  Scenario: show vessel inside map area
    Given vessel "Seal" at position "52.01N, 3.99E"
     When I view the map area between "52.10N, 3.90E" and "51.90N, 4.10E"
     Then I should see vessel "Seal" at position "52.01N, 3.99E"

  Scenario: vessels outside the map area should not be visible
    Given vessel "Seagull" at position "51.97N, 4.12E"
     When I view the map area between "52.10N, 3.90E" and "51.90N, 4.10E"
     Then I should not see vessel "Seagull"
```

Although the BDD process describes how the stories can be prioritized, it does not describe how these priorities are captured. Fortunately Scrum allows us to keep track of priorities in the product backlog.

Find scenarios

The first four stories were considered candidates for the first iteration. The tester used the stories and input from the analyst to add concrete examples to these stories. An example of a story with examples is provided in listing 4.6. For most stories the examples were easily found, and the scenario format was clear and concise. However, visual aspects like the shapes of certain markers could not be easily expressed in the textual format.

For the *Vessel Speed* story shown in listing 4.7, it was difficult to express the requirements in the form of scenarios. While there are 4 different situations (stationary, slow, regular and fast vessels), the boundary conditions between these situations are difficult to express concisely using examples.

This story also shows the problem of expressing visual properties of the user interface in scenarios, in this situation the length of the speed lines. While the tester thinks in terms of a pixel as unit length, this unit has no meaning to the customer. The tester was therefore forced to use relative lengths. This was possible because the exact lengths would have little impact on how the story was implemented. The developers could therefore estimate the implementation effort without the exact length. In situations where the visual aspects of requirements have a larger influence on the implementation the textual specification format of scenarios would not be sufficient.

4.3.3 Lessons learned

We have seen that overall the BDD process provided clear instructions for the inception phase of this project. The *Define vision and outcomes* activity went well, and most requirements could be captured in the form of features and stories without any issues.

Listing 4.7: Initial scenarios of Vessel Speed story

```
Feature: Vessel Speed
  In order to gauge the speed of vessels
  As a coast guard
  I want to see a line behind each marked vessel indicating speed

  Scenario: regular movement
    Given vessels with speeds:
        | name       | speed |
        | Sea Lion   | 10.0  |
        | Seagull    | 20.0  |
        | Seal       | 30.0  |
     When I view the map
     Then I should see speed lines:
        | name       | relative length |
        | Sea Lion   |  1.0            |
        | Seagull    |  2.0            |
        | Seal       |  3.0            |

  Scenario: slow vessels have a minimum line length
    Given vessels with speeds:
        | name       | speed |
        | Sea Lion   |  1.0  |
        | Seagull    |  5.0  |
        | Seal       | 10.0  |
     When I view the map
     Then I should see speed lines:
        | name       | relative length |
        | Sea Lion   |  1.0            |
        | Seagull    |  1.0            |
        | Seal       |  1.0            |

  Scenario: fast vessels have a maximum line length
    Given vessels with speeds:
        | name       | speed |
        | Sea Lion   | 30.0  |
        | Seagull    | 50.0  |
     When I view the map
     Then I should see speed lines:
        | name       | relative length |
        | Sea Lion   |  1.0            |
        | Seagull    |  1.0            |

  Scenario: stationary vessels have no lines
    Given vessels with speeds:
        | name       | speed |
        | Sea Lion   |  0.0  |
        | Seagull    |  0.99 |
     When I view the map
     Then I should see no speed lines
```

However, during the *Find features* activity the analyst had problems with the specification of requirements that could not be expressed as independent features. Some requirements result in interdependent features; others explicitly express a property of a wide range of other features. We refer to both types of requirements as cross-cutting requirements since they cut across a set of other features and can not be implemented as individual features within BDD.

Although BDD specifies that stories are prioritized, it provides no artifact to document priorities. However, the Scrum product backlog is a prioritized list of stories and the analyst used the backlog instead.

We also saw that some requirements were not easily expressed in the textual scenario format during the *Find scenarios* activity. Some requirements are inherently visual, like user interface elements. Other requirements can be documented with a few mathematical expressions, however showing these requirements through examples is cumbersome.

4.4 First iteration: implementing the Map View story

4.4.1 Requirements

The requirements for the first iteration were already determined during the project inception phase.

4.4.2 Implementation

Before implementation started the developers and testers estimated how much work the stories with high priority were. The customer then made a final decision on which stories to schedule for this first iteration using the feedback from the team. The customer selected the *Map View* story. Since the estimated time to complete this story was one week, this was the only story that was scheduled for this iteration.

Make acceptance test executable

This was the first technical activity, and therefore the first time that there was an opportunity to make decisions on technical aspects of the project such as the platform and tools that would be used.

After some research the developer decided to use Cucumber [3] for the automation of acceptance tests. As we described in section 4.2.1, Cucumber uses a DSL named Gherkin to enable automation using step definitions: each step of the scenarios is automated by associating blocks of code with regular expressions. When a step matches a regular expression, the block of code is executed.

In the Map View story (listing 4.6) we see four different steps on lines 7, 8, 9 and 14; the steps on line 12 and 13 are similar to lines 7 and 8, respectively. The developer therefore created four step definitions:

- `Given vessel "Seal"at position "52.01N, 3.99E"`
  Since the system depends on the AIS as the source of vessel information, this step

definition needed to encode an AIS message with the specified position and inject it into the system through the same interface that would accept AIS messages from other sources. The developer needed to determine how the message would be injected and what vessel information was needed. In this service-based system the tasks would be initiated through a service call, and therefore decisions had to be made about the service platform. The result was the following step definition:

```
Given /^vessel "([^\"]*)" at position "([^\"]*)"$/ do |name, coords_str|
  @vessel = Vessel.new Vessel::CLASS_A
  @vessel.name = name
  @vessel.position = LatLon.from_str coords_str
  transmitter = Service::lookup('ais/transmitter')
  transmitter.send_position_report_for vessel
end
```

This step definition shows a number of design decisions. The Vessel and LatLon classes represent domain concepts, while the Service class encapsulates logic of the service platform. The type of message that should be encoded depends on the class of AIS equipment used by the vessel, which was therefore added to the Vessel object.

- `When I view the map area between "52.10N, 3.90E"and "51.90N, 4.10E"`
  This step required control over the user interface in order to focus the visible map on a specific area. The Ruby code used to control the user interface depends on the type of user interface, which the developer therefore had to determine at this time. Due to the graphical nature of maps, there were two choices: a native GUI or a web interface. The developer decided to use a web interface since an extensive library was available for web-based mapping (OpenLayers [6]). Also, tool support for automated acceptance testing is better for web interfaces, since Cucumber has plug-ins for integration with browser automation tools.

  The step definition opens the map page in the browser, passing the coordinates of the area that should be shown as parameters:

```
When /^I view the map area between "([^\"]*)" and "([^\"]*)"$/ do |coords1_str, coords2_str|
  @browser = Selenium::WebDriver.for :firefox
  point1 = LatLon.from_str coords1_str
  point2 = LatLon.from_str coords2_str
  @browser.navigate_to "/?area=" << URI.encode(point1) << "," << URI.encode(point2)
end
```

  This step definition shows how browser automation is used to open the web page. In addition to the decision to use a web-based interface the developer also made design decisions about the URL and the name of the argument.

- `Then I should see vessel "Seal"at position "52.01N, 3.99E"`
  This step can be considered a test oracle: it checks whether the system works as expected by comparing the output against the expected output. In this scenario that

means verifying that a marker representing the vessel is visible at the correct location on the map.

Markers are graphical elements, and it was therefore difficult to check for the presence of a marker directly. The developer assumed the OpenLayers API would reliably show markers. The presence of a marker did then not have to be verified visually: the developer could use Javascript to retrieve a list of all markers using the OpenLayers API and check if the marker representing the vessel was included. This verification could be added to a Javascript helper function, `haveMarkerAtLatLon`, which was called using browser automation:

```
Then /^I should see a vessel at position "([^\"]*)"$/ do |point_str|
  position = LatLon.from_str point_str
  marker = @browser.execute_script('return haveMarkerAtLatLon(' <<
      position.lat << ',' << position.lon << ')')
  marker.should eq 'true'
end
```

The implementation of the helper function was deferred to the Design and implement scenario activity.

- `Then I should not see vessel "Seagull"`
  This step is very similar, and also functions as a test oracle. Since only one vessel was added and it should not be visible, there should be no visible markers. The developer decided to introduce another Javascript test helper function to check for the presence of markers:

```
Then /^I should not see vessel "([^\"]*)"$/ do |name|
  marker = @browser.execute_script('return haveMarkers()')
  marker.should eq 'false'
end
```

  Again, the implementation of the helper function was deferred to the next activity.

With the completion of these step definitions the developer had turned the scenarios into acceptance tests that could be executed with Cucumber.

Since all development was driven by the acceptance tests, the tests had to cover all architectural layers of the system where possible. The tests therefore simulated input and checked the output of the system. Our detailed description of this activity shows that it required understanding of the interfaces of the system and the messages they expect and produce. Although the BDD process does not mention design as a part of this activity, the developer was forced to design the interfaces with both the user and other systems.

Design and implement scenario

Before starting with the implementation of the first scenario, the developer had to select unit test tools to support TDD. For the specification of Ruby classes the developer chose RSpec (section 4.2.2), for Javascript the developer selected Jasmine (section 4.2.3).

Listing 4.8: Example of a RSpec specification for the `TransmitterProxy` class. The `double` function returns a mock object to which expectations are added with the `should_receive` method. If the mock object does not receive this message, the test will fail.

```
require 'spec_helper'

module Service
  describe TransmitterProxy do
    it "sends position reports to the Transmitter service" do
      vessel = "Vessel"
      socket = double('Socket')
      socket.should_receive(:send_string).with(Marshal.dump(vessel))

      t = Service::TransmitterProxy.new(socket)
      t.send_position_report_for vessel
    end
  end
end
```

The developer now proceeded with the implementation of the first scenario of the *Map View* story. Following the steps as outlined in 3.1, the developer started with the first step of the first scenario, injecting an AIS message into the system. The developer applied TDD and created a specification (listing 4.8), added functionality to the system to make the test pass and refactored the code. This was repeated until enough functionality had been added to make the system pass the first step of the scenario. This only required making the interface callable: no AIS messages were encoded or transmitted yet.

After running the acceptance tests to verify that the system did indeed pass the first step of the scenario, the developer proceeded with the second step. This step required the developer to start working on the web interface. The developer decided to use the Rails framework to create this interface, since Rails is well-documented and supports integration with the Cucumber, RSpec and Jasmine. Unfortunately the first iteration ended before the second step could be completed, and the developer was not able to deliver the story.

4.4.3 Lessons learned in the first iteration

The developer was not able to deliver the story during the first iteration: the estimation made at the start of the iteration was incorrect.

The implicit design decisions that were part of the *Make acceptance tests executable* activity made the activity difficult to complete and time-consuming.

However, most time was spent learning new technologies and tools. The developer was not familiar with Ruby and the Ruby ecosystem. The developer also had to set up a development environment by installing Ruby and the required libraries and tools and learn how the dependencies between these components are managed within a Ruby project.

4.5 Second iteration: doubling up to complete the Map View story

4.5.1 Requirements

Find features

The analyst found that no adjustments to the requirements were necessary: the features remained as listed in table 4.3.2.

Find stories

Since the developers were not able to deliver the first story, Map View, in a single iteration, the analyst and tester tried to reduce the scope of this story. However there was no clean way to split the story while still delivering business value. They therefore decided to leave the story as it was.

Prioritize stories

The customer had no changes to the priorities, the *Map View* story remained the story with the highest priority.

Find scenarios

Since no new stories were introduced and no stories were delivered, no additional scenarios were needed.

4.5.2 Implementation

Before implementation started the developers and testers again needed to estimate how much work the stories with high priority were. The Map View story was not completed in the previous iteration, and was now estimated at two weeks. It would have been preferable to split the story into multiple stories, however as discussed in 4.5.1 this was not possible. The developers and testers therefore agreed to complete the story in two weeks, effectively merging the second and third weeks into one iteration.

Make acceptance test executable

There already was an executable acceptance test for the *Map View* story, therefore no work had to be done for this activity.

Design and implement scenario

The developer resumed work at the second step of the first scenario, shown in listing 4.6. After updating the web-based user interface with an embedded map, the system was able to show a map of a specific area.

Listing 4.9: Jasmine specification for `addMarker`

```
describe("Map", function() {
        var map;
        var marker;

        beforeEach(function() {
                var latlon = new LatLon(52, 4);
                map = new Map('map', latlon);
                marker = new Marker(new LatLon(52, 4));
        });

        it("allows adding markers", function() {
                map.addMarker(marker);
                expect(map.markerLayer.markers.length).toBe(1);
        });
});
```

The third and final step of the scenario was to verify the presence of a marker on the correct location. While the previous two steps were mainly used for setting up the acceptance test, the third step would function as the test oracle, and would be the main driver for the implementation of this scenario.

The developer focussed on making the system pass the acceptance test and started at the user interface, designing and implementing Javascript code necessary to add markers to the map. First, a specification was written for an `addMarker` method (listing 4.9), then code was added to implement the `addMarker` function specification and make the unit test pass, and the code was refactored. The refactor step lead to the creation of additional units of code (e.g. `LatLon` and `Marker` classes), which were again designed and implemented using TDD.

With `addMarker` completed, the developer proceeded with the next unit of code, which would retrieve a list of markers and use the `addMarker` function to add them to the map. This procedure was repeated, each time expanding the system by implementing a unit that would use the existing units to deliver the correct system output: a marker at the correct location. By focussing on making the system pass the acceptance test and iteratively adding the functions necessary to make that happen, the developer naturally followed the outside-in design strategy outlined for this activity in section 3.4.3. We can further illustrate this by listing the first few steps during the design and implementation of the *Map View* story:

1. A marker needs to visible on the map,
   therefore a marker needs to be added to the map.

2. A marker needs to be added to the map,
   therefore marker information needs to be loaded.

3. Marker information needs to be loaded,
   therefore marker information needs to be published.

Listing 4.10: Updated scenario of Map View story

```
Scenario: show vessel inside map area
  Given class "A" vessel "Seal" at position "52.01N, 3.99E"
    And class "B" vessel "Seagull" at position "52.0N, 4.0E"
   When I see the map area between "52.10N, 3.90E" and "51.90N, 4.10E"
   Then I should see a vessel at position "52.01N, 3.99E"
    And I should see a vessel at position "52.0N, 4.0E"
```

4. Marker information needs to be published,
   therefore vessel information needs to be converted to marker information.

5. Vessel information needs to be converted to marker information,
   therefore vessel information needs to be retrieved.

While the BDD process specifies *when* the developer designs the system, it does not help the developer with the structure of the design. Since one of the influencers of BDD is DDD, DDD is an obvious candidate to help form the architecture of the system. We therefore applied the DDD design principles outlined in section 2.3.2. The developer introduced a domain layer in the system, and built the services on top of this layer. This resulted in two Ruby modules: one for domain classes and one for services using those domain classes. For example, the Vessel and LatLon classes described in 4.4.2 where moved to the Domain module, while the Service class was moved to the Service module.

The developer iteratively developed the system using TDD with the outside-in design strategy, and refined the architecture during these iterations. The changes also lead to changes to the interfaces of the system. For example, in the initial step definitions the developer had forgotten to consider how services were started or stopped. These types of changes lead to changes in the way acceptance tests were executed.

The implementation also lead to more insight into the domain, which lead to changes in scenarios and acceptance tests. For example, in the scenarios no distinction was made between class A and class B equipment. During the implementation the developer noticed that each type of equipment sends different messages, and after a discussion with the customer the tester expanded the scenarios to explicitly include both class A and class B equipment, as shown in 4.10. The developer then proceeded updating the acceptance tests and finishing the implementation.

After outside-in development had been completed, the relevant units had been implemented and passed their unit tests. However, the acceptance test did not pass. Since the acceptance test validates the system as a whole, it did not give a good indication of *where* in the system the problem occurred. The developer inspected the values passed at interfaces between units to find out where problems occurred. Most of the problems that were discovered were integration issues caused by mismatches between expectations of the interfaces: unit tests used mock objects, and these mock objects did not reliably represent the objects used in the system, often after the collaborator had been changed without updating the mock object.

Another problem was caused by concurrency issues: since each service ran inside a separate process and did not signal when it was fully started, some services were started in the incorrect order causing messages to be lost.

After the developer solved these problems and the system passed the acceptance test, the first scenario of the *Map View* story could be considered done. The second scenario required very little work, since most functionality was already in place.

4.5.3 Lessons learned in the second iteration

The developer was able to deliver the first story after three weeks. During the second iteration the developer was able to focus more on the development itself, since the initial setup of the project had been done in the first iteration.

We described in detail how the acceptance test drove the development using a outside-in design strategy, starting from the user interface and working deeper into the system. We also noted that though the BDD process does not specify how the developer should design the system, DDD can be used to provide guidelines. We also reported that insights gained during the implementation phase lead to redefining scenarios as the developers and testers gained more domain knowledge.

The main issue we encountered was the dependence on acceptance tests to verify integration. The outside-in design and strategy of BDD relies on unit tests that use mock objects as collaborators. The main cause of integration problems was a mismatch between mock objects used in unit tests and collaborators used in the acceptance test. Although the system was small a number of integration problems were discovered, and it took significant effort to find root causes with only a failed acceptance test to guide debugging. Integration could be verified earlier by using the collaborators in the unit test, however this is incompatible with the outside-in design strategy of BDD. The alternative is to add fine-grained integration tests, a practice not described as part of the BDD process.

4.6 Third iteration: delivering the second story

4.6.1 Requirements

Find features

The analyst discussed the current set of features with the customer. This did not lead to any new features, and the features remained as they are listed in table 4.3.2.

Find stories

Since no new features were added and all existing features were already documented as stories, there was no work to be done during this activity, and no new stories were created.

Prioritize stories

The customer did change the priority of the stories. Looking at the system in its current state, the customer decided to make the *Vessel Details* story a high priority, and moved it to the top of the list. The other stories kept the same relative priority.

Find scenarios

The *Vessel Details* story now had highest priority, therefore the scope of this story needed to be clarified by adding scenarios. The tester determined examples of detailed vessel information. Using the knowledge about class A and class B equipment from the previous iterations, the tester included two scenarios: one for each type of equipment. For example, the scenario for vessels with class A equipment became:

```
Scenario:
  Given vessel "Sea Lion" with details:
     | MMSI     | 245000000    |
     | Class    | A            |
     | Type     | Cargo        |
     | Position | 51.99N, 4.05E |
     | Heading  | 290          |
     | Speed    | 13.1         |
   When "Sea Lion" sends a position report
    And "Sea Lion" sends a voyage report
    And I select vessel "Sea Lion" on the map
   Then I should see all details of vessel "Sea Lion"
```

4.6.2 Implementation

The developers and testers estimated that the *Vessel Details* story would take one week, and it would therefore be the only story implemented during this iteration.

Make acceptance test executable

The scenario example given in 4.6.1 shows that new step definitions were required. While this activity cost a lot of time during the first iteration, it was now straight forward:

- The developer now had experience with the type of design decisions that had to be made.

- The interfaces used in this story were slight variations of those in the previous story, the new step definitions were therefore very similar to existing step definitions.

- The test oracle did not have to assess the presence of a graphical element such as a marker: it could verify the presence of text snippets.

The developer therefore had no problems making the acceptance tests executable.

Design and implement scenario

This activity proceeded much like during the previous iteration. The new story required mostly small modifications of the existing system. The developer added support for two more AIS message types and refactored the system in response, and focussed on the user interface aspects like loading and showing the vessel information when markers were clicked.

When the acceptance tests failed the tools did not always provide sufficient information to find the source of problems easily. The developer therefore added extensive logging: when the acceptance test failed the developer would use the log files to trace what data was exchanged between different parts of the system and pinpoint where this data deviated from what was expected.

Despite some difficulties with integration, the problems encountered during this iteration where smaller than during earlier iterations and the developer was able to deliver the *Vessel Details* story.

4.6.3 Lessons learned in the third iteration

We described how some problems surfaced while the *Vessel Details* story was implemented. The requirements phase of this iteration did not present any problems. Since this story required only smaller changes, the design decisions made during the *Make acceptance tests executable* activity where minor, and therefore did not take much time. During the *Design and implement scenario* activity the developer coped with the coarse-grained nature of the acceptance tests by introducing extensive logging to analyze integration problems.

4.7 Fourth iteration: implementing similar stories

4.7.1 Requirements

Find features

The analyst discussed the completed and remaining features with the customer, however no new features were added.

Find stories

Since no new features were added and the scope of the existing features was limited, no new stories had to be created.

Prioritize stories

The customer decided to keep the priorities of the remaining stories as it was on the original list in table 4.3.2. The candidate stories for the next iteration therefore became *Vessel Heading*, *Vessel Speed*, *Vessel Class* and *Vessel Type*, in that order.

Find scenarios

Since a story was delivered last iteration, scenarios needed to be added to some of the stories with lower priorities. In this case the scenarios were already added to the first three stories during the project inception phase, therefore only a scenario was added to the *Vessel Type* story.

4.7.2 Implementation

The developers and testers estimated the *Vessel Heading*, *Vessel Speed*, *Vessel Class* and *Vessel Type* stories at 1, 3, 1 and 1 days, respectively. They therefore committed to delivering only the first three stories.

Make acceptance test executable

Since these stories were similar to the previously implemented stories, the process of making the acceptance tests executable was roughly the same as in section 4.6.2. The main difference was that these stories did the test to verify the presence of visual elements. The combination of the heading and class of a vessel were represented by different marker shapes, while the lines were drawn separately. The confirmation of the presence of these visual properties was, like in the *Map View* story, deferred to Javascript helper functions.

Design and implement scenario

Since the stories under development use vessel information that was already introduced throughout the system by the *Vessel Details* story in the previous iteration, the changes were restricted to the user interface. Most time was therefore spent writing specifications and implementations for Javascript code. The *Vessel Speed* story took most work since it required custom rendering code to show the speed line for each marker.

4.7.3 Lessons learned in the fourth iteration

The fourth iteration was uneventful since the changes were limited to only the top layer of the architecture: the user interface. The estimations of the developers and testers were conservative, therefore one day was still remaining in this iteration after the stories had been completed. The team decided to take this time to also deliver the *Vessel Type* story by performing both the *Make acceptance test executable* and *Design and implement scenario* activities for this story. As was the case with the other stories the change was restricted to the user interface, and implementation was therefore straight forward.

Listing 4.11: The *Vessel Compliance* feature was added during the fifth iteration. Only one of the scenarios is shown for brevity

```
Feature: Vessel Compliance
  In order to see how reliable vessel information is
  As a coast guard
  I want to see which class A vessels do not comply with the AIS protocol

  Scenario: dynamic information of moving, non-anchored vessels not
      changing course
    Given non-anchored class "A" vessels with dynamic information:
        | name       | speed |
        | Sea Lion   |  1.0  |
        | Seal       |  1.0  |
        | Seagull    | 14.1  |
        | Seahorse   | 14.1  |
        | Sea Otter  | 23.1  |
        | Seahawk    | 23.1  |
     When these vessels send a position report
      And send another position report after:
        | name       | interval |
        | Sea Lion   | 10.0     |
        | Seal       | 10.1     |
        | Seagull    |  6.0     |
        | Seahorse   |  6.1     |
        | Sea Otter  |  2.0     |
        | Seahawk    |  2.1     |
     Then the compliance of the vessels should be marked as:
        | name       | compliant |
        | Sea Lion   | yes       |
        | Seal       | no        |
        | Seagull    | yes       |
        | Seahorse   | no        |
        | Sea Otter  | yes       |
        | Seahawk    | no        |
```

4.8 Fifth iteration: changing requirements

4.8.1 Requirements

Find features

The analyst discussed the system and the remaining features with the customer. The analyst found an additional feature during this activity, the *Vessel Compliance* feature shown in listing 4.11.

Find stories

The analyst did not consider it necessary to split the *Vessel Compliance* feature into multiple stories, and therefore the feature was used as a story. The other remaining story, *Vessel Movement* also had a sufficiently small scope.

Prioritize stories

The customer decided to give the new story a high priority, which meant that the *Vessel Compliance* story would be implemented before the *Vessel Movement* story.

Find scenarios

The tester now added scenarios to the two remaining stories. The scenarios for the *Vessel Movement* story were not challenging. However, the *Vessel Compliance* story was not as trivial. The compliance of a vessel to the AIS protocols is determined by the interval with which the vessel sends certain types of AIS messages. The expected interval also depends on message type, speed, changes in heading and the navigation status. A different interval is associated with each combination of these factors. Eventually the tester decided to create a different scenario for each (message type, navigation status) pair, and to create create different steps to handle the remaining combinations inside the individual scenarios. An example is shown in listing 4.11.

4.8.2 Implementation

Both the new *Vessel Compliance* story and the *Vessel Movement* were estimated at one week. Since the *Vessel Compliance* story had higher priority, the developers and testers committed to delivering only this story.

Make acceptance test executable

The step definitions of the scenarios were complex, which caused this activity to take more time than during the previous few iterations. However, no new issues were discovered with this activity.

Design and implement scenario

The raw AIS feed links provides a timestamp for each message. For the previous stories this information was not relevant, therefore the system discarded these timestamps when the messages entered the system. Since the *Vessel Compliance* story depended heavily on the timestamps, the system now had to accept and propagate the timestamps with all AIS-related messages sent throughout the system. Many parts of the system had to be changed as a result. As before, the developer started with the changes at the user interface, then worked inward from there, service by service. The developer also added a ComplianceService service that determined compliance based on the frequency of incoming messages. Although the logic behind the service was quite complex, the story was finished in time.

4.8.3 Intermezzo: crashing demo

When the system was demonstrated to the customer the system crashed. Dynamic AIS messages hold the speed of the vessel, however vessels can transmit a special value to indicate that the speed is unknown. Acceptance tests did not simulate this edge condition,

and while some unit tests took this situation into account, other parts of the system assumed vessels to always report a value for speed. This integration issue caused the crash.

Although a scenario should have been created that included this situation, it is likely that some scenarios will be initially overlooked in projects. The automated acceptance tests are therefore not always sufficient to get a stable system.

The crashing demo also brought to light that the process of BDD does not define how bugs are handled. A bug that is discovered in the same iteration as the implementation of the related story should be addressed in that iteration. However, when a bug is found later, as is the case here, this is not possible.

4.8.4 Lessons learned in the fifth iteration

During this iteration the customer introduced new requirements. The requirement influenced many parts of the system, however the design changes were determined in small steps by applying the outside-in design strategy. The BDD process was therefore capable of handling the new requirement, and the story could be delivered on time.

However, a serious bug was found in the system during the demonstration to the customer. This showed that the automated acceptance tests are not always sufficient to provide a stable system: an edge condition may not have found its way into a scenario. The discovery of a bug outside the iteration also lead to the insight that the BDD process does not specify how to handle bugs.

4.9 Sixth iteration: finishing the project

4.9.1 Requirements

Find features

The analyst and customer discussed whether features were missing from the system. The analyst found no new features.

Find stories

Only the *Vessel Movement* story remained, and the scope of this story was limited enough for the story to be implemented in one iteration.

Prioritize stories

With only one story left prioritization was not necessary.

Find scenarios

Scenarios had already been added to the *Vessel Movement* during the previous iteration.

4.9.2 Implementation

The developers and testers estimated the *Vessel Movement* story to take 3 days, and committed to delivering this story.

During the demonstration at the end of the previous iteration a bug was discovered. Since that demonstration and the requirements phase of this iteration took place during the same session, the developer fixed the bug immediately after that session, which is effectively during the implementation phase described here.

Make acceptance test executable

The scenarios were very similar to earlier scenarios, and making the acceptance test executable showed no issues.

Design and implement scenario

The impact of this story was again limited to the user interface. Rather than loading a list of markers once, the web interface had to be adapted to reload this list every second. Using TDD the developer made the appropriate changes, and the developer was able to deliver the story on time.

4.9.3 Intermezzo: crashing demo

During the demonstration, the browser stayed open showing the system while a discussion went on, and the system eventually crashed. Since acceptance tests only exercised the system for a short amount of time, the developers and testers had never noticed that the system was opening new network sockets faster than they were released. Since the system was running longer during the demonstration, this caused a crash.

4.9.4 Lessons learned in the sixth iteration

The sixth iteration did not present challenges until the final demonstration to the customer, when the system crashed again. Although this was originally meant to be the final iteration, the customer decided to extend the project with one iteration in order to resolve this issue.

4.10 Seventh iteration: fixing the last bug

The only remaining issue was the bug, however BDD does not propose how to handle such bugs. The developer therefore proceeded to fix the bug outside the process. First, the root cause of the issue was determined. The developer then proceeded to use TDD to implement the changes necessary to fix the bug.

4.10.1 Lessons learned in the seventh iteration

We were not able to use the BDD process, and did therefore not gain insight into the BDD process beyond the lack of support for bug fixing, which we already noted during the previous iterations.

4.11 Discussion

In our detailed description of iterations of the project we have run into a number of issues. In this section we will describe observations that are relevant to an evaluation of the BDD process as it was used in the case study, yet were not tied to specific iterations.

4.11.1 Deploying the system

One observation relates to the deployment of the system. At the end of each iteration the product should be stable, and the customer should be able to use it to deliver the business value documented in the features and stories. Before the customer can use the system, however, the system needs to be deployed.

Any system has requirements: they require external libraries or tools, specific operating systems or specific hardware. During the case study this caused problems. When we tried deploying the system on another machine we discovered that the system had implicit dependencies on system libraries. In the BDD process there is no indication of how these requirements should be documented. Even if deployment is highly automated, the automation will still expect a specific base system.

4.11.2 System architecture

The BDD process suggests that all design activities take place in the TDD cycle of the *Design and implement scenario* activity. However, we have seen that the developer uses system interfaces to create acceptance tests during the *Make acceptance tests executable* activity, which implies earlier design decisions.

Since developers perform all design activities individually, developers make the design decisions themselves. In the case study only one developer worked on the system and this was not an issue. However when more developers are involved it is difficult to build a coherent architecture without guiding principles.

Although BDD promotes the use of the outside-in design strategy, it does not offer suggestions about how the architecture of the system should be organized. We suggest using DDD as outlined in section 2.3.2; domain concepts are modelled as classes in a domain layer of the system. We applied this in the case study, and added all domain classes to a `Domain` package in Ruby. Our further design activities were driven by the service-oriented architecture imposed as a cross-cutting requirement.

Although the low-level design can be derived from the code of the system, the architecture is not documented. When new developers are added to the team, or developers return to the project after an extended period of time, they will have to review the low-level design to relearn the high-level model of the architecture. This can be time-consuming for

large systems, especially if the developer is only interested in making a specific change to the system and then continue work on another system. The design will also not show *why* the system was designed this way: the assumptions and justifications for decisions are not documented. As the system evolves, developers can not clearly see when these assumptions and decisions are no longer appropriate, and it becomes difficult to refactor the system.

4.11.3 Domain knowledge

One aspect of BDD is the use of domain language through-out the project: in conversations with the customer, in requirements and in code. Therefore all participants need an understanding of the domain and the domain language. However, BDD does not propose how to share this information.

For example, if the customer introduces the AIS-related term MMSI during the *Find features* activity, the analyst will learn the meaning of MMSI from the customer, and may use this in a feature description. When the tester joins the process for the *Find scenarios* activity, the customer will again have to outline what an MMSI is. Similarly, when a developer has to implement the scenario, the term MMSI is unknown. The situation is exasperated when a person joins the team at a later iteration, and the person needs to be brought up to speed on many domain concepts.

Although BDD stresses the use of the domain language suggested by DDD, we suggest going further and adopting the design philosophy as well. DDD suggests that the design of the system should model the domain, which could help new developers learn concepts. Additionally, a document showing a model of the core domain concepts and their relations could aid these new developers.

Chapter 5

Extending the BDD process

5.1 Introduction

In the previous chapter we described how we discovered some parts of the process that were missing or incomplete. We will use this chapter to clarify and add to the process description we outlined in chapter 3, extending the BDD process as is was described in literature. The issues we encountered were:

- During the first iteration a significant amount of time is spent on setting up a development environment.
- Not all types of requirements can be expressed as stories; in particular requirements that cut across multiple stories and requirements that relate to the user interface presented problems.
- The process does not give any guidance for working with bugs.
- All design decisions are made by developers individually and the architecture of the system is not documented.
- The *Automate acceptance tests* activity involves a lot of design decisions with regards to the user interface and interfaces with other systems, while the description of the activity does not mention these aspects.
- The automated acceptance tests were not always successful at discovering issues.
- The delivery of the system is not handled, and there is no artifact documenting how the system is deployed.

For each phase of the process we describe in detail the related issues and how we address these issues. We also describe the new or updated artifacts, roles and activities in full detail. We only describe the changed aspects of the process, parts of the process described in 3 not explicitly modified should be considered part of this extended process.

5.2 Project Inception

During the first iteration of the case study project the developer spent a lot of time on learning new technologies and tools and setting up a development environment. It is difficult to schedule these activities in the same iteration as the initial set of stories: since the time required for these activities is hard to estimate, the team may end up delivering none of the stories.

Since each iteration should be geared towards delivering business value it would be more appropriate for the developers and testers to learn about the tools and prepare the development and testing environments during the project inception. We suggest facilitating this by extending the process of BDD by adding an activity *Set up development environment*, which is similar to what is described as the exploration phase in Extreme Programming in [14].

5.2.1 Activities

Set up development environment

The *Set up development environment* activity allows developers and testers to set up initial development and testing environments and get familiar with tools and technologies required for the project. While some of the technologies depend on the architecture and will be chosen during iterative development, others are known in advance based on the first high-priority requirements.

In a BDD process likely tools to be explored during this activity are acceptance- and unit testing tools. Developers can also learn more about the programming environment that will be worked with, the associated best practices, build- and dependency management and setting up a version control system. The output of this activity should be development- and testing environments that allow the team to start delivering business value as soon as the first iteration starts.

In the context of the case study the developer would use this time to learn Ruby, how dependencies are managed in Ruby projects and the test tools Cucumber and Rspec. Learning to work with Jasmine would not have been part of this activity, since it was not clear that Javascript would be used until the first iteration.

5.3 Requirements

During the gathering of the initial requirements we noted that some requirements can not be implemented as independent features since they require functionality that cuts across sets of other features. We also found that visual aspects of requirements, for example the user interface, could not easily be expressed in the textual format of scenarios. When visual aspects have a large impact on the effort required for its implementation, the story becomes hard to estimate.

We suggest adding a *Find cross-cutting requirements* activity, where the analyst determines the requirements that cut across a set of features, which makes it explicit that the

Table 5.1: An example of a cross-cutting requirements specification, with two cross-cutting requirements.

Request Logging	In order to determine service usage As a system operator I want all service requests to be logged
Service monitoring	In order to be able to use the existing service monitoring infrastructure As a system operator I want all services to communicate using the ZeroMQ protocol

analyst needs to find this type of requirements. The requirements are added to a list, the *Cross-cutting requirements specification* artifact. Testers and developers reference this list during subsequent activities to ensure that the appropriate tests are created and the implementation satisfies all cross-cutting requirements.

For stories with visual requirements we propose a *user interface prototype* artifact, as a high-level description of the user interface. This prototype is not a complete specification, and communicates by its form: the prototype is sketched by hand or created with a wireframe prototyping tool. Since user interface design requires a skill set that differs from existing roles, we also introduce a new role, *User interface designer*.

In later iterations of the case study project we also found that the process of BDD does not specify how bugs should be handled. We suggest documenting each bug in a *Bug report* and finding the root cause of the bug in the *Investigate bug report* activity. Adding regression tests and fixing the bug is deferred to later phases of the process.

5.3.1 Artifacts

Cross-cutting requirements specification

Cohn [16] suggests adding cross-cutting requirements to so-called *constraint cards*, since they often describe constraints of the system, however [16] does not indicate when the cards are referenced. Adzic [10] describes a checklist for these types of requirements, which is referenced every time testers and developers start working on a feature to see which requirements impact the current story.

We suggest extending the idea from [10] by using a checklist of cross-cutting requirements where each requirement is specified in the feature format. An example of such a *Non-functional requirements specification* is given in table 5.3.1.

User interface prototype

For projects with high risk associated with the user interface [16] proposes the use of paper prototypes. [10] adds that wireframe prototyping tools are more suitable when prototypes need to be shared.

We propose an *user interface prototype* for capturing non-trivial changes to the user interface. The prototype is appended to the story that drives the change, and serves two pur-

poses: to enable more accurate estimates of implementation effort, and to guide developers during design and implementation.

Like stories, the user interface prototype is not a complete specification and to communicate this clearly the prototype should not look completed. This can be achieved by giving it an unfinished look through the use of wireframe prototyping tools or by sketching the user interface by hand. Figure 5.1 shows an example of a UI prototype for the *Vessel Speed* feature from listing 4.7.

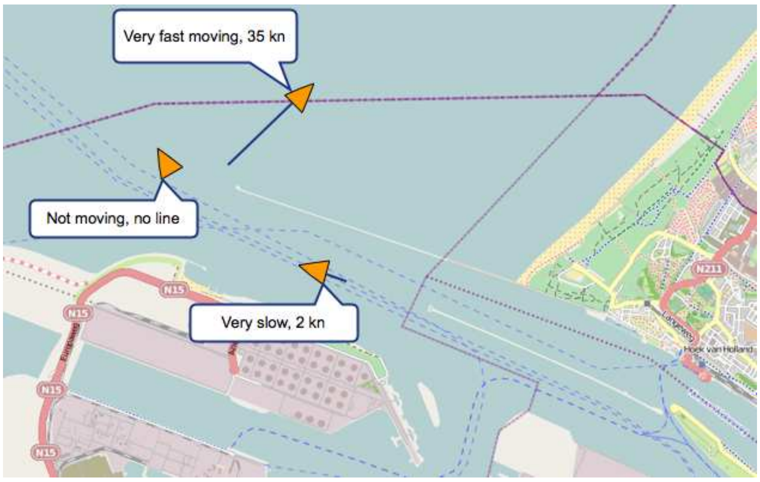


Figure 5.1: An example of an user interface prototype created with a wireframing tool, for the feature *Vessel Speed* shown in listing 4.7.

Bug report

Following the suggestion of Cohn [16], we suggest adding a *Bug report* with the feature format. The steps needed to reproduce the bug are added as one or more scenarios. These scenarios are added immediately when the bug report is created, and are refined as a developer investigates the bug to find the root cause with the *Investigate bug report* activity. An example of a bug report is shown in listing 5.1.

The *Bug report* can be prioritized and implemented as if it were a story. However, unlike a story, a bug report does not have to be entirely written in domain language. The story and its scenarios represent an issue with the system and can therefore contain language used in any domain used in the system, including technical language.

5.3.2 Roles

User interface designer

Finding the correct user interface and capturing the interactions in a prototype requires a special set of skills. We emphasize this by adding the role of user interface designer. In addition to drafting the visual requirements in *User interface design* artifacts, the *User*

Listing 5.1: Example of a bug report for the final bug of the case study, discovered during the sixth iteration.

```
Feature: Reduce Number Of Sockets Used
  In order to be able to run the system for a long period of time
  As a system administrator
  I want the system to use a limited number of sockets

   Scenario: run system a long time
     When I view the homepage
      And wait 1 hour
     Then the system should still run
```

interface designer assists the developer during the implementation of stories to ensure an adequate user interface for the system.

5.3.3 Activities

Find cross-cutting requirements

The analyst identifies requirements that can not be implemented as independent features, and instead influence how other features are implemented. These cross-cutting requirements are documented in the feature format documented in 3.2.1. The resulting requirements specifications are added and maintained on a separate list, the *cross-cutting requirements specification*.

Prioritize stories

In addition to stories describing new changes, we now have bug reports describing bugs found in features delivered in previous iterations. These bug reports are treated as if they were stories, however the analyst and customer should consider bugs to have a relatively high priority. The longer a bug remains in the system, the higher the chance that it will cause additional problems and reduce development speed.

Create user interface prototype

The user interface designer reviews the stories with a high priority and considers the required changes to the user interface. When the changes are not trivial, the designer investigates the best way for the user role to interact with the system, in cooperation with the analyst and the users. This results in a rough specification of the user interface documented in a *User interface prototype*.

Investigate bug report

The scenario of the bug report in listing 5.1 is not suitable as an acceptance test; a single test run would take more than an hour. The tester does not know the root cause of the issue, and can therefore not reduce the scope efficiently with scenarios. The developer will therefore

use this activity to investigate the root cause of issues described in bug reports with a high priority. The developer then refines the scenarios that the tester added when the bug report was created.

For example, for the bug report of listing 5.1, the root cause was found to be that the web server opened too many connections to the vessel service. A more suitable scenario to use as a regression test was:

```
Scenario: show map a number of times
   When I view the homepage
    And wait 3 seconds
   Then there should be only one connection to the vessel service
```

5.4 Design

In our description of the BDD process we added the *Make acceptance test executable* activity to the implementation phase of iterative development. However, our experiences during the case study (section 4.4.2) showed that design is an important part of the activity: acceptance tests can not be made executable without an understanding of the interfaces of the system, which requires designing those interfaces. To emphasize the design aspect we suggest moving the activity to the design phase.

In our discussion in section 4.11.2 we discussed the need for an overview of the architecture of the system. Evans [18] suggests the use of documents and diagrams to guide people to central points of the design. We suggest adding a light-weight document for this purpose. This document, the *Architecture description*,is updated during the *Update architecture description* activity, where an *Architect* leads a team discussion about high-level design decisions. The activity has the added benefit of forcing developers to align their views of the architecture and cooperate on design changes.

5.4.1 Artifacts

Architecture description

The *Architecture description* outlines a high-level view of the system, and is updated throughout the project. The goal is to keep the views of the system that different developers have aligned and allow developers joining the project later to get an idea of the structure of the system.

The description contains a high-level textual description of the architecture and an overview of the components, their interconnections and their responsibilities using a combination of graphical and textual representations. The *Architecture description* should also show the project file organization to clarify where the components live within the source code.

The exact form of the *Architecture description* depends on the architecture of the system. For a service-oriented system, like the project in the case study, we suggest representing services and their interconnections in the form of a UML component diagram, with a short description of each service to highlight the responsibilities. An example is shown in

appendix A. For other systems a similar representation with dependencies between packages or modules can be used, for example in the form of a UML package diagram and a short description of each package.

The low-level design details should not be part of the *Architecture description*, since these details are most accurately captured by the code of the system. Therefore a listing or description of individual classes or files is generally not appropriate.

5.4.2 Roles

Architect

The architect leads the developers in the creation or update of the *Architecture description*. The role of architect will usually be filled by a senior developer, and different developers may take this role during different iterations. Together with the developers and testers the architect is responsible for the architecture of the system.

5.4.3 Activities

Adjust architecture

The architect leads a session where the architecture changes are discussed that are required for the stories that will be implemented. This is a group effort involving the architect, the developers and the testers. The goal is to create and maintain a coherent view of the high-level design of the system, whereas the low-level design is deferred to the implementation phase.

The developers assess which changes to the architecture are necessary for each story of the current iteration, while the testers assess testability. If the changes are non-trivial and affect the high-level functions of the system, the *Architecture description* is updated.

Make acceptance tests executable

To emphasize the design aspects of this activity it has been moved to the design phase. Additionally, developers should consider the task of designing the system interfaces an explicit first step for this activity.

The bug reports discussed in section 5.3.1 have scenarios representing the steps used to reproduce the bug. Those scenarios are also made executable during this activity and function as acceptance- and regression tests.

5.5 Implementation

We discussed the challenges we encountered with the *Make acceptance test executable* activity in the previous section. There were no other issues with the implementation phase, however changes elsewhere in the process do influence how the *Design and implement scenario* activity is performed.

5.5.1 Activities

Design and implement scenario

The TDD cycle itself did not present any problems during the case study. However we have some suggestions that add more detail to the description this activity in a BDD process.

Before the developer starts working on a scenario, the developer reviews the cross-cutting requirements documented in the *Cross-cutting requirements specification*. The requirements from the checklist that impact the current scenario are taken into account when making design changes. If a *User interface prototype* is available for the story with which the scenario is associated, the developer will use the prototype as a guideline when implementing changes that affect the user interface.

5.6 Testing

In our case study we saw that the system crashed during the demonstrations at the end of the fifth and sixth iterations. These problems were not discovered earlier because during acceptance tests the system would generally only run for a short period of time, and with a restricted set of input data. We address this problem by adding an additional testing activity, *Acceptance testing*.

5.6.1 Activities

Acceptance testing

The tester uses this activity to do extensive acceptance testing. The tester runs the system for a longer period of time and explores manually whether the functions outlined by the scenarios indeed work correctly. The tester can also use larger data sets as input for the tests.

When the tester discovers problems related to stories that are worked on during the current iteration, the developers will fix these problems as soon as possible. If an issue is discovered related to functionality implemented during earlier iterations, the tester documents the issue in a *Bug report*, after which the customer can prioritize it appropriately.

5.7 Delivery

As we outlined in section 4.11.1, the BDD process does not offer any guidelines for the deployment of a system. All systems have dependencies, and these should be documented. Since the team responsible for deployment is often not the same team that develops the system, instructions are also required to explain how the system can be deployed successfully.

5.7.1 Artifacts

Deployment instructions

The deployment instructions explicitly state which dependencies the system has. This includes the operating system and external libraries, with version numbers where appropriate. An UML deployment diagram can be used to provide this information efficiently for systems that are distributed over multiple machines.

Additionally, the deployment instructions describe the process of installing and configuring, starting and stopping the system and include instructions for maintenance tasks that need to be performed on the system once it has been installed. When the project focusses on extending an existing system the deployment instructions should be extended with upgrade instructions.

Finally, it may be appropriate to add instructions for running the tests associated with the system. These instructions allow others to assess whether deployment was successful.

5.7.2 Roles

Developer

The developers are responsible for updating the deployment description during or at the end of each iteration.

5.7.3 Activities

Update deployment instructions

The developer reviews the changes that have been made to the system and updates the *Deployment instructions* to reflect the latest state of the system. When possible the developer verifies the instructions by executing them on a clean machine.

5.8 Summary

In this chapter we have suggested a number of modifications and extensions of the original BDD process outlined in chapter 3. We have updated table 3.6 to reflect these changes; the result is shown in table 5.8.

Table 5.2: An overview of the extended BDD process listing the artifacts, roles and activities of each phase. New artifacts, roles and activities have been emphasized.

Project Inception	
Artifacts:	Vision Statement
Roles:	Analyst, Tester, *Developer*
Activities:	Define vision and outcomes, Define initial requirements, *Set up development environment*
Requirements	
Artifacts:	Feature, *Cross-cutting requirements specification*, Story, Scenario, *User interface prototype*, *Bug report*
Roles:	Analyst, Tester, *User interface designer*, *Developer*
Activities:	Find features, *Find cross-cutting requirements*, Find stories, Prioritize stories, Find scenarios, *Create user interface prototype*, *Investigate bug report*
Design	
Artifacts:	*Architecture description*, *Acceptance test*
Roles:	*Architect*, *Developer*
Activities:	*Adjust architecture*, *Make acceptance tests executable*
Implementation	
Artifacts:	Unit test
Roles:	Developer
Activities:	Design and implement scenario
Testing	
Artifacts:	None
Roles:	*Tester*
Activities:	*Acceptance testing*
Delivery	
Artifacts:	*Deployment instructions*
Roles:	*Developer*
Activities:	*Update deployment instructions*

Chapter 6

Continuing the case study

In chapter 5 we clarified and modified the process description of the BDD process based on our experiences during the case study. However, without an assessment of these modifications it is unclear whether they have a positive effect on the process. We therefore continued the case study for one more iteration, using the modified process description.

We will use this chapter to describe the second part of the case study using the same structure we used for the first part, described in chapter 4. For each role we describe the experiences during each activity, including how the activity was performed and difficulties that were encountered, guided by question 2 of 1.1.

We continued the project from first part of the case study, therefore there was no project inception phase. Since we used a modified process with additional artifacts, we created those artifacts to represent the state of the system just before the iteration started. We will describe these artifacts in the following section, after which we will report on the first and only iteration of the project in section 6.2. In section 6.3 we will discuss the lessons we learned from this second part of the case study.

6.1 Preparing for the first iteration

Before we started a new iteration with the modified process, artifacts that were not part of the original process had to be created. New artifacts, roles and activities are emphasized in table 5.8; in this section we will describe how these artifacts were created.

6.1.1 User interface prototype

Since each *User interface prototype* is tied to a specific story, and all existing stories had been completed, it was not necessary to create any prototypes.

6.1.2 Bug report

No known bugs remained from the previous iterations, therefore no *Bug report* artifacts had to be created.

Listing 6.1: The *Cross-cutting requirements specification* is used to document requirements that are not tied to a specific story.

```
In order to leverage existing knowledge
As the customer
I want the system to be programmed in the Ruby programming language

In order to make it easy to interoperate with the system
As the customer
I want the system to consist of loosely coupled services communicating using the ZeroMQ protocol

In order to prevent license costs for mapping
As the customer
I want to use Open Street Map for mapping

In order to get up-to-date vessel information
As a coast guard
I want the vessel information to be updated with information from the AIS
```

6.1.3 Cross-cutting requirements specification

As we described in section 4.3.2, the system did have a number of cross-cutting requirements which could not be documented during the first part of the case study. During our preparations for the next iteration we added a text file to the project with the cross-cutting requirements in the form of a checklist, as described in section 5.3.3. The checklist is shown in listing 6.1.

6.1.4 Acceptance test

Although the *Acceptance test* artifact is new to the design phase of the iteration, the same acceptance tests were already used during the implementation phase of the original project. Therefore, acceptance tests had already been created for each of the existing stories and no additional acceptance tests had to be created.

6.1.5 Architecture description

The architecture of the project had been documented only in the source code. Our next iteration also required us to have a document outlining the architecture at a high level. During our preparations we created such a document, shown in appendix A.

The document starts with a high level description of the logical layers of the system architecture. We also added a structural overview of the system. In our case study the system had a Service Oriented Architecture, therefore one of the key components was an overview of the services in the form of a UML component diagram (figure A.1). Table A.2 provides a short description of each service to clarify its responsibilities.

The logical layers and structural properties of those layers should be reflected by the directory structure of the source code when possible. However, sometimes other factors

influence this directory structure. In our project we used the Rails framework for the user interface, and Rails has certain conventions with regards to the directory layout of projects. Additionally, our project consisted of both Ruby and Javascript source code. Therefore the directory structure did not follow directly from the architecture, and we added a short overview of all significant directories in table A.2.

6.1.6 Deployment instructions

The developers had not yet created a document with instructions for the deployment of the system. Such a document should outline the system requirements, installation procedure, configuration, and how the system is started and stopped. The developers created this document, shown in appendix C, before continuing with the next iteration. The developers were already using the *Bundler* tool to manage the dependencies between Ruby libraries; these dependencies were therefore already listed in a file within the project. This file is referred to from the deployment instructions.

6.2 First and only iteration

In this section we will describe the experiences of each process role during a single iteration of the modified process. For each phase of the iteration we describe how each role performed the activities and what difficulties were encountered.

6.2.1 Requirements

Find features

During the discussion with the customer, the analyst found one new feature for the system:

```
Feature: Area Speed Compliance
  In order to see vessels that do not comply with area rules
  As a coast guard
  I want to see which vessels move too fast in an area with speed limits
```

Find cross-cutting requirements

The analyst did not find any additional cross-cutting requirements. Therefore the cross-cutting requirements specification did not need to be updated.

Find stories

Since there was only one feature, it was the only candidate for this iteration. The analyst discussed the details of the *Area Speed Compliance* feature with the customer. This lead to the insight that the feature involved two aspects: defining and displaying areas with speed limits, and marking vessels that do not comply with those limits. The analyst therefore split the feature into the two stories shown in listings 6.2 and 6.3.

Listing 6.2: The *Area Speed Definition* story, with one example scenario

```
Feature: Area Speed Definition
  In order to see if vessels are in an area with a speed restriction
  As a coast guard
  I want to define an area with a speed restriction

  Scenario: define an area
    Given I see the map area between "52.10N, 3.90E" and "51.90N, 4.10E"
     When I define an area with a maximum speed of "19.0" knots and
         coords:
        | 52.00N, 3.97E |
        | 52.02N, 3.98E |
        | 52.00N, 3.99E |
      And I refresh the page
      And I see the map area between "52.10N, 3.90E" and "51.90N, 4.10E"
     Then I should see the area with restrictions marked on the map
```

Listing 6.3: The *Area Speed Compliance* story, with one example scenario

```
Feature: Area Speed Compliance
  In order to see vessels that do not comply with area rules
  As a coast guard
  I want to see which vessels move too fast in an area with speed limits

  Scenario: vessel inside area, moving too fast
    Given class "A" vessel "Seal" at position "52.01N, 3.98E"
      And vessel "Seal" has a speed of "25.0" knots
      And an area with a maximum speed of "15.0" knots and coords:
        | 52.00N, 3.97E |
        | 52.02N, 3.98E |
        | 52.00N, 3.99E |
     When I see the map area between "52.10N, 3.90E" and "51.90N, 4.10E"
     Then vessel "Seal" should be shown as non-compliant
```

Prioritize stories

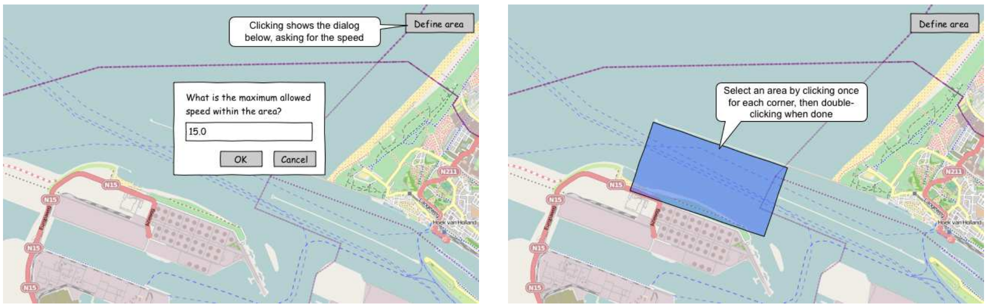
The customer prioritized the *Area Speed Definition* story over the *Area Speed Compliance* story, since the latter did not have much utility without some way to define areas with speed limits.

Find scenarios

The tester proceeded to add scenarios to the two stories with the input from the customer. Listings 6.2 and 6.3 show one scenario per story, as an example; the other scenarios were slight variations of these examples.

Create user interface prototype

With input from the customer the UI designer attempted to find an efficient way of interacting with the system for the user stories with high priority, which now includes both new user stories. The UI designer walked through the actions the user would perform to define an area with a speed limit, and how the system would give feedback. This flow was documented in UI prototypes. For our project we chose to sketch the significant steps of this flow using a firewrame tool; these sketches are shown in figures 6.1 and 6.2.



(a) First the user enters the speed limit in a dialog box.

(b) Then the user selects the area where the limit should be applied by clicking the points that form the corners of the area

Figure 6.1: The UI prototype for the *Area Speed Definition* story consisted of a series of wireframes that show how the user chooses the speed limit within the area and then selects the area on the map.

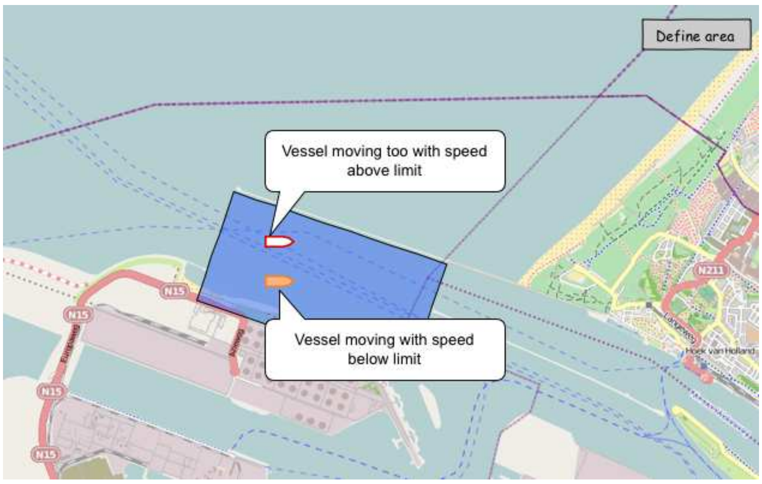


Figure 6.2: The UI prototype for the *Area Speed Definition* story consisted of a single wireframe showing how compliant and non-compliant vessels are marked on the map.

Investigate bug report

There were no bugs remaining from previous iterations, therefore no bug reports had to be investigated and this activity was not performed during this iteration.

6.2.2 Design

At this point the developers and testers had to estimate the amount of work required to complete the two stories. They estimated that the *Area Speed Definition* story would take 4 days, and the *Area Speed Compliance* story could be completed in 1 day. Since both stories could be completed in a single iteration, both stories were selected for this iteration.

Adjust architecture

During this activity the architect should lead a discussion with the developers about the high-level changes to the design of the system. Since in this case study the architect and the developer roles are taken on by the same person, the architect made the decisions himself.

The new functionality for these stories included the definition and retrieval of areas and verification of compliance of individual vessels. Taking into account the existing system and the *Cross-cutting Requirements Specification*, the architect decided to add this functionality to the system in the form of a new service, the `AreaComplianceService`. The architect then updated the architecture description by updating the service diagram and adding a description for the new service. Appendix B shows the updated *Architecture description* artifact.

Make acceptance tests executable

During the first part of the case study all the interfaces of the system had to be designed during this activity. However, now the main decisions about the user interface were already documented with the *Create user interface prototype* activity, and the interfaces with other systems already received attention during the *Update architecture description* activity. The developer was able to leverage the work done during those earlier activities when creating the step definitions necessary to make the acceptance tests executable.

The developer still had to make some design decisions with regards to the testability of the system. For the *Area Speed Definition* the user selects an area by selecting the corners of the area on the map. This situation is similar to what was discussed in section 4.4.2 with regards to the map markers: directly simulating the area selection flow was difficult. The developer solved this the same way as in section 4.4.2. The step definition called a test function that would trigger the area selection event with the area specified by a function argument. Now the step definition only had to pass the coordinates provided in the scenario as arguments to the call of the test function. The test function itself was implemented during the *Design and implement scenario* activity.

6.2.3 Implementation

Design and implement scenario

The developer started working on the first story, *Area Speed Definition*, using the outside-in design strategy with TDD. The developer started with the design and implementation of the user interface functions, then moved on to the web layer and finally added the `AreaCompliance` service with functions to store and retrieve areas.

The developer followed the same procedure for the *Area Speed Compliance* story. However, the architect did not foresee all necessary changes to the architecture, which manifested itself in the service diagram. The `AreaCompliance` service receives AIS messages that need to be decoded using the `PayloadDecoder` service. However, during the *Update architecture* activity the architect did not add that connection to the service diagram. The developer chose to add that connection during the current activity.

6.2.4 Testing

Acceptance testing

When the tester ran the system for a longer period of time, the tester discovered an aspect of the *Area Speed Compliance* story that was not covered by a scenario. When a vessel entered an area with a speed over the limit, it was marked as non-compliant. However, when the vessel left this area, the vessel was not marked as compliant again. Although no scenario covered it, a discussion with the customer revealed that the vessel should indeed be marked as compliant in such a situation. The tester then created an additional scenario for this situation. The issue was considered a bug, and the scenario was implemented within the same iteration.

The tester also discovered a bug that caused the system to accept only one area at a time. This issue was not discovered with the automated test since the scenarios only defined one area at a time. This was also fixed within the same iteration.

6.2.5 Delivery

Update deployment instructions

The developer introduced a dependency on an external Ruby library, and added this dependency to the dependency file within the project source code. Since the deployment instructions already reference this file for the Ruby dependencies, the instructions themselves did not need to be updated, and remained as they are shown in appendix C. The developer also performed the installation procedure on a virtual machine. Although this was somewhat time consuming, it did show that the procedure worked correctly.

6.3 Lessons learned

During this iteration case study project continued with an updated process. Since there was no project inception phase, the *Set up development environment* activity was not performed.

There were also no bugs from previous iterations that needed to be fixed, therefore the *Investigate bug report* activity was not required.

The introduction of the *Find cross-cutting requirements* activity did not have an effect on the rest of the process. This may have been due to the fact that the number of cross-cutting requirements was limited and already well known from the previous iterations.

The *Create user interface prototype* activity gave the testers and developers extra information about the user interface before they had to estimate the effort required to complete the stories. It also made the later *Make acceptance tests executable* activity easier, since the prototype gave the developer an clear idea of the user interface.

While a group discussion should be part of the *Adjust architecture* activity, this was not possible in our case study since only one person was involved. Therefore the effect of this additional communication could not be evaluated. The *Architecture description* artifact that resulted from the activity was only referenced a few times, to review the connections between services. However the artifact turned out to be incomplete.

During the iteration the developer discovered that some changes required by the stories had not been foreseen during the *Adjust architecture* activity. Also, at the end of the iteration the developer discovered one service was not shown in the *Architecture description* even though it was part of the system. Both discrepancies show the difficulty of creating and maintaining an accurate *Architecture description*, even for a small project. Linking the description more directly to the source code, either through static analysis, annotations or coding conventions, may be able to help automate the creation of the description, which in turn makes it trivial to keep the artifact up-to-date.

Manual acceptance testing during the *Acceptance testing* activity helped discover an issue with the implementation before the system was demonstrated to the customer, and was therefore a valuable addition to the process.

Using the *Deployment instructions* created and updated during the *Update deployment instructions* activity it was possible to install the system on another machine. These instructions should therefore allow anybody with access to the source code of the system [8] to install the system.

Chapter 7

Discussion

In our problem statement in section 1.1 we outlined a number of questions to guide our research. In this thesis we have described the work done to answer these questions. We will now, for each question of the problem statement, summarize and discuss the results from the previous chapters. We will also report the threats to the validity of these results in section 7.1.

1. *What is the software engineering process of Behavior-Driven Development?*

   We used existing literature to research the software engineering process of Behavior-Driven Development. By combining information from a number of different sources we were able to outline the process as a set of artifacts, roles and activities.

2. *Does the process provide sufficient information to complete a project?*

   To evaluate whether the software engineering process of Behavior-Driven Development provides sufficient information we conducted a case study. We completed one project consisting of seven iterations using the process description we compiled from our literature research. One person took on the role of the customer, while another person took on all other roles. During each iteration of the project we considered the three questions:

   a) *Is it clear what artifacts look like, and do the artifacts provide enough information for stakeholders to perform the activities?*

      We found that requirements that cut across stories and visual requirements like the user interface can not be expressed as stories, and can therefore not be documented as part of the process. Also, the process does not describe how bugs or the deployment of the system should be documented, and there is no artifact outlining the architecture of the system.

   b) *Are the roles of all involved stakeholders described accurately?*

We found that existing roles are described accurately. However, since developers generally do not have strong user interface design skills, we found that a specific role for this expertise is missing from the process.

c) *Is it clear what each activity encompasses, and do the stakeholders have sufficient information to perform the activities?*

The *Make acceptance tests executable* activity is not clearly described. This activity requires knowledge of the interfaces of the system, which are not documented at that point in the process. The developer therefore has to design these interfaces during the activity, while this is not part of the description of this activity.

We also discovered tasks that need to be performed that are not part of the documented activities. During the first iteration a lot of time was spent setting up and learning tools, while this is not part of an activity. Also, the process lacks activities to handle bugs discovered in the system, and there is no activity for additional testing.

Overall our case study showed that gathering and specifying requirements that fit the story format works well, as does the Test-Driven Development cycle of implementing stories. However, outside these two areas the process is unclear and incomplete.

3. *If the process does not provide sufficient information, how can we clarify or extend the process description?*

The results of the case study show a number of areas where the original process can be extended to provide more guidance. We chose a number of additional artifacts, roles and activities that could contribute to a more complete process description.

Finally, we evaluated some of these elements by applying the extended process to a single iteration of the project. We found that the addition of a *Create user interface prototype* activity helped the developer with estimation and reduced effort required for later activities. The *Adjust architecture* activity was not as effective; although it helped the developer prepare for later activities, the artifact representing the architecture was not correct and quickly outdated. The *Acceptance testing* activity added an explicit testing activity for the tester, and was successful in finding issues that were not discovered with the automated acceptance tests.

7.1 Threats to validity

7.1.1 Internal validity

One person took on all roles of the team, and therefore the communication between roles was likely not be taken into account sufficiently. We tried to mitigate this by consciously evaluating situations as if other people were involved while performing the activities, however it likely has had an effect on our case study.

Also, we suggested some changes that we did not evaluate in the final iteration of the case study. Although we have motivated the changes, their effects on the process could not be gauged. A more complete evaluation is left as future work.

7.1.2 External validity

The developer was not yet familiar with the Ruby programming language and the tools used during the case study. The effort required to set up a development environment may not be representative of other projects, since developers may already be familiar with the technologies and tools involved in their project. However, even if the associated effort is lower, projects are likely to include some setup activities such as configuring source code management tools, build environments and continuous integration servers.

The case study consisted of a single project, which also leads to a number of external threats to validity.

- The project involved a web application, our findings may therefore not be applicable to other types of applications such as desktop applications, mobile applications or real-time systems. More case studies are required to determine whether this is the case.

- The system had a Service Oriented Architecture, and our findings may not be applicable to systems with other architectures due to the incremental nature of design activities. In our case study the services formed one layer of the architecture, future work has to be done to show whether the results hold for layered architectures in general.

- The system was built using specific technologies (i.e. Ruby, Rails, Javascript) and tools (Cucumber, RSpec, Jasmine). When working with technologies that do not have similar tools it may not be possible to follow the process, or it may require significantly more effort. However, BDD-style acceptance and unit testing tools already exist for a wide range of technologies. The tools are also not complex, the effort required to port them is therefore limited.

7.1.3 Personal observations

As a participant in the process our experiences were instructive and generally positive. By strictly following the process we were forced to think about each activity and its outcomes, which we think had a positive impact on how activities were performed.

The stories and scenarios provide a clear reference for all roles. Acceptance Test-Driven Development and Test Driven Development provide an easy to follow cycle of design and implementation and give direct feedback about progress, which is satisfying for testers and developers. However, as developer we experienced the extensive use of mock objects as a downside; when behavior of a class changes, modifications have to be made in three places: in the unit test, the implementation and the mock object representing the class in other unit tests.

The architecture of the system was somewhat different than we had expected before the case study started. The choice the developer made for the Rails framework turned out to add unnecessary complexity. Rails ended up only providing a thin layer between the services and the Javascript user interface, a task for which much simpler Ruby servlets could have been used. On the other hand we had expected the vessel information to be stored in a database; the developer chose to store the information in a hash table instead, a simpler solution.

Overall our experiences with the software engineering process of BDD lead us to believe it is an especially good candidate for teams with limited experience with processes, since it provides a good sense of progress and uses light-weight artifacts.

Chapter 8

Conclusions and future work

8.1 Conclusions

We determined and described the software engineering process of Behavior-Driven Development, and applied that process to a project during a case study. The results of the case study show that the process is incomplete: we found that the process does not facilitate cross-cutting and visual requirements, bug handling and documentation of the architecture and deployment procedure. We also found that automated acceptance testing is not sufficient to deliver a stable system.

We suggested process changes that mitigate or solve these issues, which lead to a modified process description. In the second part of the case study we evaluated a number of our suggested modifications. Our effort to improve the documentation of the architecture was ineffective, since the documented architecture quickly diverged from the implementation. A suitable method for handling this problem is left to future work.

However, the results show that most of the evaluated changes are an improvement of the process. Additional activities and specification formats for cross-cutting and visual requirements lead to clearer requirements, and additional testing was successful in preventing the delivery of an unstable system. We conclude that a modified process that includes these changes gives us a more complete software engineering process for Behavior-Driven Development.

8.2 Contributions

We have made the following contributions:

- We provided a structured description of the software engineering process of Behavior-Driven Development.
- We evaluated the process in terms of clarity and completeness by applying the process to a project.

- We suggested a number of changes to artifacts and activities that clarify the process and make it more complete, and performed a preliminary evaluation that showed which changes improve the process.

8.3 Future work

- *More complete evaluation of extended process*
  We have made a number of suggestions for changes to the process, however only a part of these suggestions were part of the final evaluation. For example, the artifacts and activities related to project inception, bug handling and deployment have not been evaluated. One possibility for future work is therefore a full case study that applies the process with all changes to a complete project.

- *Documenting architecture*
  The second part of the case study showed the issue of keeping the documentation of the architecture synchronized with the implementation. Future work could involve research that links the documentation directly to the implementation, for example through annotations in the source code.

- *Project types*
  We have applied the software engineering process of Behavior-Driven Development to one application. The type of application can be characterized as a web-application with a Service Oriented Architecture. Our evaluation and subsequent suggestions may give different results when applied to other types of projects, such as desktop applications or real-time systems. Future research could evaluate the process in a number of projects of varying types.

- *Alternative artifacts*
  A number of artifacts have been added, such as the *User interface prototype*, *Architecture description* and *Deployment instructions*. We have tried to keep these artifacts simple and straight-forward, in line with agile philosophies. However, other formats may be more suitable, and suitability may depend on certain properties of projects.

- *Improved tooling*
  Stories and scenarios directly link individual requirements to the implementation. We think this connection can be exploited to provide tools that give developers more information about the system. For example, by combining acceptance tests with coverage information the developer can gain insight into the parts of the system involved in fulfilling a requirement, which is valuable information when behavior has to be changed. It may also be possible to leverage this information to provide an indication of the quality of the design.

Bibliography

[1] Behat: BDD for php. `http://behat.org`.

[2] behave is behaviour-driven development, python style. `http://packages.python.org/behave/`.

[3] Cucumber: Making BDD fun. `http://cukes.info/`.

[4] Manifesto for agile software development. `http://agilemanifesto.org/`.

[5] Mocks aren't stubs. `http://martinfowler.com/articles/mocksArentStubs.html`.

[6] OpenLayers. `http://openlayers.org/`.

[7] OpenStreetMap. `http://www.openstreetmap.org/`.

[8] Source code of case study ais project. `https://github.com/ais-case/ais`.

[9] Zeromq. `http://www.zeromq.org/`.

[10] Gojko Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications, 2011.

[11] Johan Andersson, Geoff Bache, and Peter Sutton. XP with acceptance-test driven development: A rewrite project for a resource optimization system. In Michele Marchesi and Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, pages 1011–1011. Springer Berlin / Heidelberg, 2003.

[12] Dave Astels. A new look at test-driven development. `http://techblog.daveastels.com/files/BDD_Intro.pdf`.

[13] Kent Beck. *Test-Driven Development By Example*. Addison Wesley, 2002.

[14] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

[15] David Chelimsky, Dave Astel, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2010.

[16] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004.

[17] Torkild Eriksen, Gudrun Høye, Bjørn Narheim, and Bente Jensløkken Meland. Maritime traffic monitoring using a space-based AIS receiver. *Acta Astronautica*, 58(10):537 – 549, 2006.

[18] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2003.

[19] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337 – 342, 2004.

[20] IMO. New and amended performance standards. Technical Report MSC.74(69), IMO, 1998.

[21] ITU-R. Technical characteristics for an automatic identification system using time-division multiple access in the VHF maritime mobile band. Technical Report ITU-R M.1371-4, International Telecommunicaton Union, 2010.

[22] D. Janzen and H. Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, sep 2005.

[23] D. S. Janzen and H. Saiedian. Does test-driven development really improve software design quality? *Software, IEEE*, 25(2):77–84, march-april 2008.

[24] Ron Jeffries and Grigori Melnik. Guest editors' introduction: TDD–the art of fearless programming. *Software, IEEE*, 24(3):24–30, May–June 2007.

[25] Lasse Koskela. *Test Driven: TDD and Acceptance TDD for Java Developers*. Manning Publications, October 2007.

[26] Dan North. Introducing BDD. `http://dannorth.net/introducing-bdd/`.

[27] Dan North. There's more to BDD than evolving TDD. `http://dannorth.net/2006/06/04/theres-more-to-bdd-than-evolving-tdd/`.

[28] Carlos Solís and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387, 2011.

Appendix A

Architecture description for initial case study

In this document we provide an example of a architecture description for the initial case study. This description outlines the high-level architecture and the project file layout as it was at the end of the case study described in chapter 4.

The layered architecture of the project has three main layers, from top to bottom:

1. *Web layer*
   The web layer is responsible for interaction with the user through a web interface. The web layer selectively retrieves information from the service layer and presents this information in the form of maps, icons and popups to communicate the information to the user.

2. *Service layer*
   The service layer performs the main tasks of the system: it processes the incoming AIS messages and serves the extracted information to the web layer. The service layer is kept thin by having this layer only coordinate objects from the domain layer. Each services is concerned with interaction with a limited number of domain objects and how the input and output required for the interactions is communicated with other services.

3. *Domain layer*
   The domain layer holds objects and operations that relate to the maritime/AIS and mapping domains, like vessels, AIS message encoding and AIS message types.

Source code related to each of these layers is kept in separate directories in the project files, although there are two directories for the domain layer (one for Ruby code and one for Javascript code).

The structure within the layers is also relevant. For the web layer, the system uses the default Rails (MVC) structure. The structure of the domain layer follows from the domain itself. However the distribution of the tasks performed in the service layer is more artificial, and we will therefor discuss this structure in the next chapter.

A.1 Service overview

The internal structure of the service layer does not follow naturally from the domain, and is specific to this system. Since the system is small, we can discuss individual services, the tasks they perform and their interconnections, without losing the overview. In figure A.1 the relationships between the services are shown, while a short description of each service is provided in table A.1.

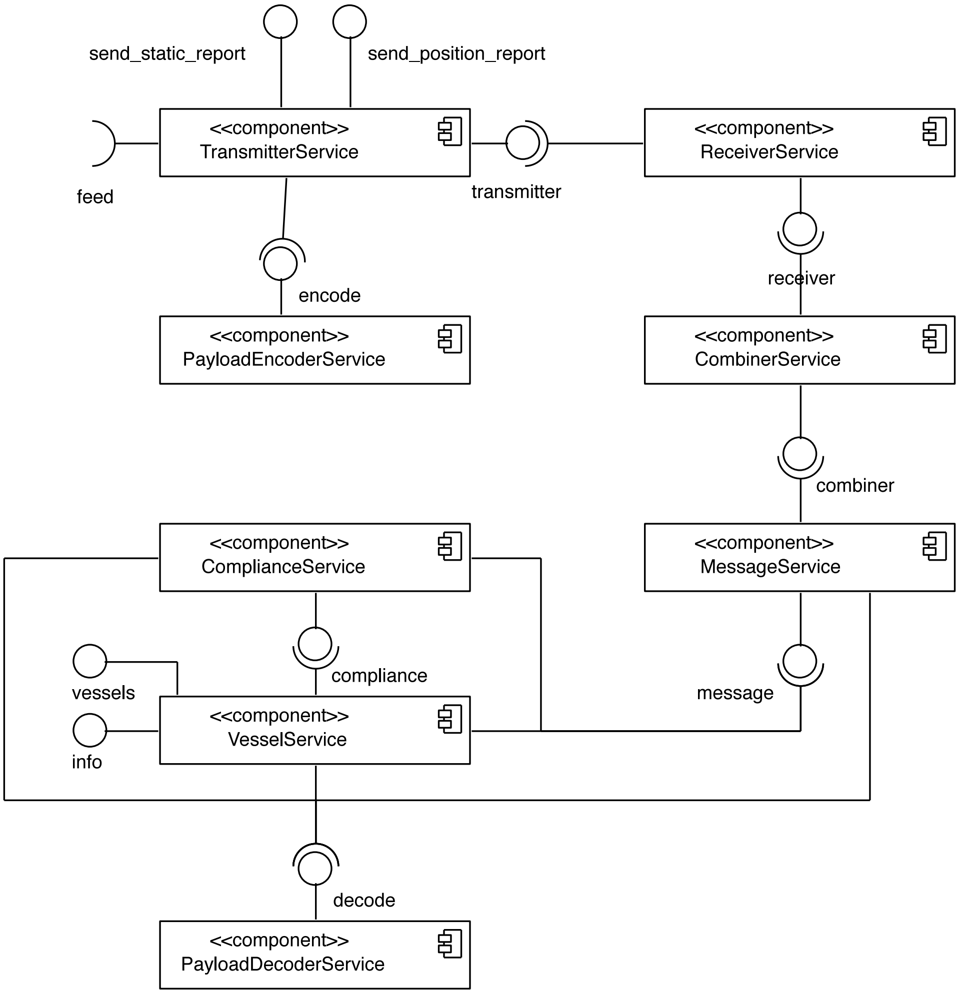


Figure A.1: An overview of the connections between services in the form of a UML 2.0 component diagram.

A.2 Project directory structure

The directory structure of the project files is outlined in table A.2. This table gives a short description of the types of files located in the most important directories of the project.

Table A.1: Short descriptions of each of the services.

CombinerService	Combines AIS fragments into complete AIS messages and publishes the complete AIS messages
ComplianceService	Tracks incoming messages to determine whether the vessel is compliant with AIS specifications related to message frequency. When a vessel becomes non-compliant, the service publishes a message with the vessel MMSI.
MessageService	Determines the message type of incoming AIS messages and republishes each message with the type as a prefix. This allows services that subscribe to the MessageService to only listen to AIS messages of specific types and filtering out other messages.
PayloadDecoderService	Decodes a raw AIS message payload into a Message object.
PayloadEncoderService	Encodes a Message object into a raw AIS message payload.
ReceiverService	Verifies the checksums of raw AIS message fragments and publishes only valid fragments
TransmitterService	This service functions as a proxy between the raw feed and the ReceiverService, and allows service clients to inject AIS messages representing position- and static reports into the feed. The TransmitterService was added to allow acceptance tests to easily inject certain types of messages without having to modify the rest of the system.
VesselService	Keeps track of the last known vessel information for all vessels using information published by the ComplianceService and MessageService. The service allows access to this information by providing interfaces to retrieve a list of all vessels or information of a specific vessel (by MMSI).

Table A.2: The directory structure of the case study project.

`app/`	Web layer, in the form of a Rails application
`config/`	Configuration files
`features/`	Features files, in Gherkin syntax
`features/step_definitions/`	Cucumber step definitions
`lib/`	Main source tree, general purpose code (logging)
`lib/assets/images`	Images that are required by the system, e.g. icons
`lib/assets/javascripts/`	Javascript source tree
`lib/assets/javascripts/domain/`	Javascript domain layer sources
`lib/domain/`	Ruby domain layer sources
`lib/services/`	Ruby service layer sources
`log/`	Log files generated by the system
`public/`	Static web files, e.g. OpenLayers themes
`script/`	Shell scripts for common command-line tasks
`spec/`	Specifications for use with RSpec and Jasmine
`spec/controllers/`	Ruby specs for Rails controllers
`spec/javascript/`	Javascript specs
`spec/lib/`	Ruby specs for classes in the lib/ directory
`spec/views/`	Ruby specs for Rails views
`vendor/`	External libraries, i.e. OpenLayers and ZeroMQ

Appendix B

Architecture description for second phase of case study

In A we provided the architecture description for the initial case study. As we proceeded with the second phase of the case study, the architect updated that description with the *Adjust architecture* activity. The result is shown below; the main change is that an Area-ComplianceService was added.

The layered architecture of the project has three main layers, from top to bottom:

1. *Web layer*
   The web layer is responsible for interaction with the user through a web interface. The web layer selectively retrieves information from the service layer and presents this information in the form of maps, icons and popups to communicate the information to the user.

2. *Service layer*
   The service layer performs the main tasks of the system: it processes the incoming AIS messages and serves the extracted information to the web layer. The service layer is kept thin by having this layer only coordinate objects from the domain layer. Each services is concerned with interaction with a limited number of domain objects and how the input and output required for the interactions is communicated with other services.

3. *Domain layer*
   The domain layer holds objects and operations that relate to the maritime/AIS and mapping domains, like vessels, AIS message encoding and AIS message types.

Source code related to each of these layers is kept in separate directories in the project files, although there are two directories for the domain layer (one for Ruby code and one for Javascript code).

The structure within the layers is also relevant. For the web layer, the system uses the default Rails (MVC) structure. The structure of the domain layer follows from the domain itself. However the distribution of the tasks performed in the service layer is more artificial, and we will therefor discuss this structure in the next chapter.

B.1 Service overview

The internal structure of the service layer does not follow naturally from the domain, and is specific to this system. Since the system is small, we can discuss individual services, the tasks they perform and their interconnections, without losing the overview. In figure B.1 the relationships between the services are shown, while a short description of each service is provided in table B.1.

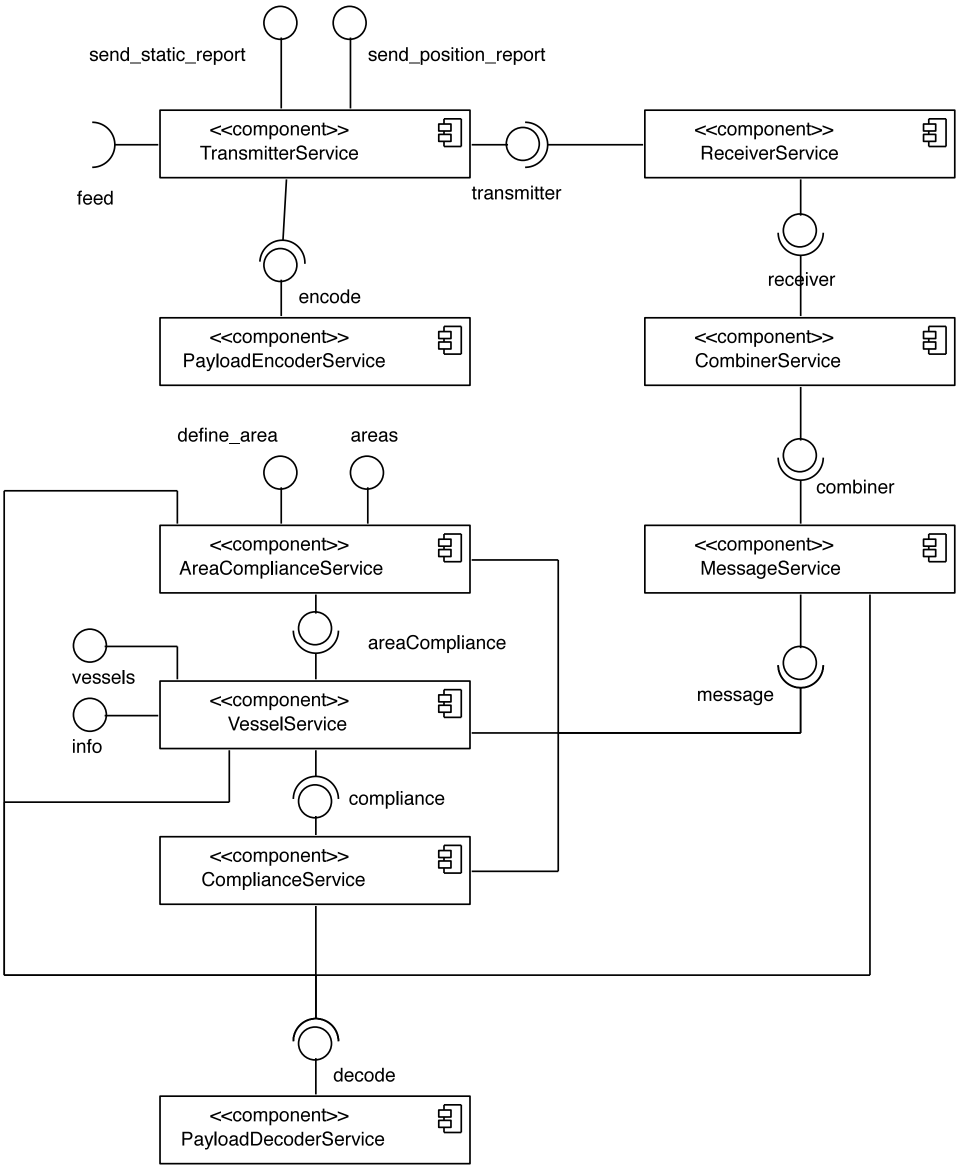


Figure B.1: An overview of the connections between services in the form of a UML 2.0 component diagram.

Table B.1: Short descriptions of each of the services.

AreaComplianceService	Allows the definition and querying of areas with a speed limit. The service keeps track of vessel speeds inside these area and publishes a message with the vessel MMSI when a vessel becomes non-compliant.
CombinerService	Combines AIS fragments into complete AIS messages and publishes the complete AIS messages
ComplianceService	Tracks incoming messages to determine whether the vessel is compliant with AIS specifications related to message frequency. When a vessel becomes non-compliant, the service publishes a message with the vessel MMSI.
MessageService	Determines the message type of incoming AIS messages and republishes each message with the type as a prefix. This allows services that subscribe to the MessageService to only listen to AIS messages of specific types and filtering out other messages.
PayloadDecoderService	Decodes a raw AIS message payload into a Message object.
PayloadEncoderService	Encodes a Message object into a raw AIS message payload.
ReceiverService	Verifies the checksums of raw AIS message fragments and publishes only valid fragments
TransmitterService	This service functions as a proxy between the raw feed and the ReceiverService, and allows service clients to inject AIS messages representing position- and static reports into the feed. The TransmitterService was added to allow acceptance tests to easily inject certain types of messages without having to modify the rest of the system.
VesselService	Keeps track of the last known vessel information for all vessels using information published by the AreaComplianceService, ComplianceService and MessageService. The service allows access to this information by providing interfaces to retrieve a list of all vessels or information of a specific vessel (by MMSI).

Table B.2: The directory structure of the case study project.

`app/`	Web layer, in the form of a Rails application
`config/`	Configuration files
`features/`	Features files, in Gherkin syntax
`features/step_definitions/`	Cucumber step definitions
`lib/`	Main source tree, general purpose code (logging)
`lib/assets/images`	Images that are required by the system, e.g. icons
`lib/assets/javascripts/`	Javascript source tree
`lib/assets/javascripts/domain/`	Javascript domain layer sources
`lib/domain/`	Ruby domain layer sources
`lib/services/`	Ruby service layer sources
`log/`	Log files generated by the system
`public/`	Static web files, e.g. OpenLayers themes
`script/`	Shell scripts for common command-line tasks
`spec/`	Specifications for use with RSpec and Jasmine
`spec/controllers/`	Ruby specs for Rails controllers
`spec/javascript/`	Javascript specs
`spec/lib/`	Ruby specs for classes in the lib/ directory
`spec/views/`	Ruby specs for Rails views
`vendor/`	External libraries, i.e. OpenLayers and ZeroMQ

B.2 Project directory structure

The directory structure of the project files is outlined in table A.2. This table gives a short description of the types of files located in the most important directories of the project.

Appendix C

Deployment instructions for case study

In this document we provide an example of a deployment description for the case study. This description outlines the as it was at the end of the case study using the BDD process as outlined in 3.

C.1 System requirements

The system requires the following software to be installed:

- Ruby 1.9.3-p194, with openssl, zlib and libyaml support
- ZeroMQ 2.2.0
- Rubygems

The system also depends on a number of Ruby gems, which are listed in the `Gemfile.lock` file of the project. These gems are managed with the bundler tool, and will be installed during the installation process outlined below.

There are no specific hardware requirements, however the hardware platform needs to be supported by the software system requirements.

C.2 Installation

Installing is as simple as downloading the project code from `https://github.com/ais-case/ais`, and installing and running bundler from the project root:

```
gem install bundler
bundle install
```

C.3 Starting the system

From the main project folder you can start the service platform:

```
rake services:start
```

The process will run indefinitely, keeping the service processes alive in the background until you kill the rake process with `CTRL-C`. To launch the web interface run the Rails web server:

```
rails server
```

You should now be able to access the web interface at `http://localhost:3000`.

C.4 Stopping the system

To stop the system kill both the rake process and the rails server with the `CTRL-C` keyboard shortcut.

C.5 Running tests

There are three separate test suites, using a variety of tools. These tools are automatically installed when you run bundle install.

Acceptance tests use the Cucumber tool, and are driven by browser automation with Capybara. Browser automation requires Mozilla Firefox to be installed. The tests are defined in the `features` subdirectory. Run the acceptance tests with:

```
rake cucumber
```

One note: acceptance tests have additional requirements, which are not installed by default. To install these additional dependencies run `bundle install --without none` from the root of the project.

Ruby unit tests use the RSpec tool, and can be found in the `spec` subdirectory. Run the unit tests with:

```
rake spec
```

Javascript unit tests use the Jasmine tool, and can be found in `spec/javascript`. To run the tests use:

```
rake jasmine:ci
```